TEC-0078

Image Understanding Environment for ARPA Supported Research and Applications

Douglas Morgan

Booz, Allen and Hamilton, Inc.
1500 Plymouth Street
Mountain View, CA 94043-1230

January 1996

Prepared for:
Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

Monitored by:
U.S. Army Corps of Engineers
Topographic Engineering Center
7701 Telegraph Road
Alexandria, Virginia 22315-3864

US Army Corps of Engineers
Topographic Engineering Center

T
E
C

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** January 1996	**3. REPORT TYPE AND DATES COVERED** Final Technical Sep. 1989 - Jul. 1995

4. TITLE AND SUBTITLE

Image Understanding Environment for ARPA Supported Research and Applications

5. FUNDING NUMBERS

DACA76-89-C-0023

6. AUTHOR(S)

Douglas Morgan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Booz, Allen and Hamilton, Inc.
1500 Plymouth Street
Mountain View, CA 94043-1230

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Advanced Research Projects Agency
3701 North Fairfax Drive, Arlington, VA 22203-1714

U.S. Army Topographic Engineering Center
7701 Telegraph Road, Alexandria, VA 22315-3864

19. SPONSORING / MONITORING AGENCY REPORT NUMBER

TEC-0078

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report documents the design and development of a Booz, Allen and Hamilton (BAH) Vision Environments system, and contributions by BAH to the design of the ARPA Image Understanding Environment (IUE). The primary contributions are to an object-oriented specification of a hierarchy of classes for image understanding applications. Analyses of implementation issues in both the C++ and Common Lisp languages are also presented.

14. SUBJECT TERMS

Image Understanding, Environments,
Object-Oriented Specification

15. NUMBER OF PAGES

244

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	Unlimited

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Preface

This report describes work performed under Contract DACA76-89-C-0023 by Booz•Allen & Hamilton, Inc., Mountain View, California, 94043-1230 for the U.S. Army Topographic Engineering Center (TEC), Alexandria, Virginia 22315-3864, and the Advanced Research Projects Agency (ARPA), Arlington, Virginia 22203-1714. The Contracting Officers' Representative at TEC is Ms. Lauretta Williams. The ARPA point of contact is Dr. Oscar Firschein.

Section 1

Introduction

This is the final technical report for project DACA76-89-C-0023, an Advanced Research Projects Agency (ARPA) contract to Booz•Allen & Hamilton (BAH) titled "An Image Understanding Environment for ARPA Supported Research and Applications." The contract was originally let to Advanced Decision Systems (ADS) which was subsequently purchased by BAH.

The project began as a team effort by BAH and the Georgia Institute of Technology (GT) to design a Vision Environment system and develop a prototype in C++ for UNIX workstations. Initial designs and a prototype were developed in the first year of the contract. Plans were also made for significant improvements to the prototype. However, at the beginning of year two, it became clear that the effort would achieve the greatest benefit to ARPA if design activities were oriented toward collaboration on the emerging ARPA Image Understanding Environment (IUE) design. The IUE design was to be a joint effort by a large segment of the ARPA Image Understanding (IU) community. To enhance the eventual impact of the IUE system, the Vision Environments design effort was consolidated into the IUE design effort. Like the Vision Environment system, the ARPA IUE is intended to facilitate the transfer of technology from the ARPA IU community into industrial, military, and commercial applications.

The focus of activity since the first year has been helping to design the ARPA sponsored IUE. BAH added additional subcontractors and a consultant to provide expertise in areas important to the wider scope of the IUE design. The additional subcontractors (with principal investigators) were Stanford University (Professor Thomas O. Binford) and SRI International (Dr. Thomas Strat). The consultant was Dr. Joseph Mundy from General Electric Corporate Research and Development.

The IUE design significantly advances the state-of-the-art in environments and development frameworks. Key advancements are capabilities to seamlessly integrate the numerous concepts of IU into one system, to easily introduce new users to the system, and to extend the system in multiple new directions. Our work has focused on providing these capabilities with a class hierarchy clearly embodying IU concepts and a user interface allowing complex interactions to be simply expressed.

The IUE was designed primarily by a group of ten ARPA IU community representatives from industry and academia. The design group consisted of:

Joe Mundy (Chair)	– GE
Thomas Binford	– Stanford
Terry Boult	– Columbia
Al Hanson	– U Mass
Bob Haralick	– U of Washington
Charlie Kohl	– AAI
Daryl Lawton	– Georgia Tech
Douglas Morgan	– BAH
Keith Price	– USC
Tom Strat	– SRI.

The design efforts by Joe Mundy and representatives from BAH, GT, Stanford, and SRI have been either partially or completely funded through this contract. Prior to the formation of the design group, several meetings (open to the ARPA IU community) were held to refine the IUE goals and development strategy. During this period, the Vision Environments design and prototyping experience was critical to focusing the IUE design on specification of a comprehensive class hierarchy.

This document describes the results obtained under this contract. Activities and results during the first two years are covered in annual technical reports included as appendices. The period from September 26, 1991 to July 25, 1993 is described in the body of the report. The focus during this period was analysis and exploration of design concepts developed by the IUE design committee.

The activities performed during this last period included

- Attending and presenting at IUE design committee meetings, including a presentation of the IUE at INRIA, France that firmly established European interest in the IUE (All).
- Updating the IUE design documents and contributing to IUE overview papers (All).
- Prototyping designs and providing example IUE application code examples (All).
- Analyzing efficiency and usability issues involved with multiple virtual inheritance in C++ (BAH).
- Suggesting subtyping hierarchies for the IUE that conform to the constraints of class derivation (inheritance) in C++ (BAH).

- Being the IUE committee liaison with the ARPA/Texas Instruments Open Object Oriented Database project (BAH).
- Prototyping base classes in CLOS (BAH).
- Improving definitions of base classes and proposing that Manifold classes augment or replace IUE Spatial-Object classes (BAH).
- Participating in defining the Lisp/C++ interface requirements of IUE and communicating those requirements to Lucid, Inc., the Lisp/C++ IUE interface ARPA contractor (BAH/SRI).
- Defining IUE user interface capabilities and helping design Spatial-Object classes (GT).
- Helping specify Image-Feature classes (Stanford).
- Chairing the IUE design committee (Mundy).
- Helping define and test IUE data exchange facilities (Mundy).
- Organizing IUE meetings and committee activities (Mundy).
- Specifying coordinate transforms and coordinate systems (SRI).

This effort resulted in and contributed to, multiple documents for general distribution. These include the IUE Overview (the "Little Blue Monster"), the IUE Class Definitions (the "Big Green Monster"), the IUE Data Exchange Manual, several ARPA image understanding workshop papers, and papers for IEEE Computer Vision and Pattern Recognition workshops. The IUE overview, design, and data exchange documents are now maintained by Amerinex Artificial Intelligence, Inc. (AAI) as part of the IUE integration contract from ARPA. The latest versions of these documents (as well as several proceedings papers) may currently be obtained over the Internet from http://www.aai.com/AAI/IUE/IUE.html.

This document describes the additional analysis and prototyping that BAH performed. Section 2 presents results of an analysis of multiple virtual inheritance in C++. Section 3 discusses how the IUE inheritance hierarchy might be reorganized to do without "dynamic attributes." Section 4 presents some of the recommendations presented to the TI Open Object Oriented Database effort. Section 5 describes the base classes prototyped in CLOS. Section 6 presents a design for mathematics-oriented classes. Parts of this design have helped improve the definitions of specific IUE classes, though the entire design has not been adopted by the IUE. Appendices contain the annual technical reports of years one and two of the contract.

Section 2

Analysis of Multiple Inheritance in C++

This section summarizes results of experiments investigating the effects of multiple virtual inheritance on object execution speed and storage size. Experiments were performed using the compilers: g++ 1.39.1, g++ 2.0, and CC 2.1. Lucid Common Lisp (which only has the equivalent of virtual bases) was also used for comparison with non-C++ inheritance techniques. The experiments set up complex inheritance hierarchies and measured instance size, creation and destruction time, slot access time, and virtual function (method) call time.

A typical test was to create an inheritance hierarchy such as in Figure 2-1. All timings were normalized to the speed of a 25MHz SPARCstation 1+. Measurements were done with and without the "-O" optimization compilation flag. The time used in calling the various virtual functions ma through md from classes a, b1, ..., g1, c5, ..., g5, d15, g15, and h were computed. If all calls are made on a variable declared to be of class a, then timings are as follows:

```
g++:            All calls take 1.04 usec
g++ -O:         All calls take 0.68 usec
g++-2.0:        All calls take 1.24 usec
g++-2.0 -O:     All calls take 0.56 usec
CC2.1:          All calls take 1.27 usec
CC2.1 -O:       All calls take 0.60 usec
```

```
                    a
                    |
+-------+-------+---+---+-------+-------+
|       |       |       |       |       |
b1      c1      d1      e1      f1      g1
|       |       |       |       |       |
|       c2      d2      |       f2      g2
...     ...     ...     ...     ...     ...
|       |       |       |       |       |
|       c5      d5      |       f5      g5
...     ...     ...     ...     ...     ...
|       |       |       |       |       |
|       |       d14-+   |       |       g14--+
|       |       |   |   |       |       |    |
|       |       |   d15 |       |       |    g15
|       |       |       |       |       |
+-------+-------+---+---+-------+-------+
                    |
                    h

Classes have no data members.  All functions are empty.
Define empty virtual functions: ma, mb, mc, md in class a
Override virtual function for mb in class b1
Override virtual function for mc in classes mc<i> , i=1,...,5
Override virtual function for md in classes md<i> , i=1,...,15
Override all virtual functions in h
```

Figure 2-1: Example inheritance hierarchy

If all calls are made on a variable declared to be of class d10, then timings are as follows:

```
g++:                All calls take 3.41 usec
g++ -O:             All calls take 1.93 usec
g++-2.0:            All calls take 3.61 usec
g++-2.0 -O:         All calls take 1.85 usec
CC2.1:

TIME (USEC) TO CALL VIRTUAL FUNCTION ON OBJECT OF CLASS

                VIRTUAL FUNCTION
C             ma   mb   mc   md
L           +-------------------
A   d10     | 1.71 1.70 1.87 1.27
S   d15     | 1.70 1.71 1.70 1.26
S   h       | 1.70 1.86 1.86 1.26

CC2.1 -O:

TIME (USEC) TO CALL VIRTUAL FUNCTION ON OBJECT OF CLASS

                VIRTUAL FUNCTION
C             ma   mb   mc   md
L           +-------------------
A   d10     | 1.25 1.27 1.26 0.61
S   d15     | 1.25 1.27 1.27 0.61
S   h       | 1.25 1.26 1.27 0.61
```

Object creation and destruction times vary widely with position of the class of interest within the inheritance hierarchy and with allocation on the stack or on the heap (using the C++ operators new and delete). The timings are:

```
  Operation              g++    g++ g++2.0 g++2.0 CC2.1     CC
                                 -O           -O            -O

10 a's on/off stack:      2.8    2.8    2.8    2.7    5.7    0.8
new/delete 10 a's:       76.1   69.4  219.9  208.5  263.7  230.5
10 d15's on/off stack:  452.4  404.3   49.4   45.8 2161.7 1413.9
new/delete 10 d15's:    541.0  483.1  273.0  251.8 2882.5 1836.0
10 h's on/off stack:   1078.9  916.4  144.4  135.6 4313.2 2522.9
new/delete 10 h's:     1442.5  996.7  370.4  343.6 5110.4 3593.1
```

The sizes of instances shows a critically important feature of these C++ implementations. C++ object system overhead causes instances to grow large with number of virtual base classes. Measurements show such significant size increase that the IUE cannot reasonably make virtual derivation the environment standard. Size measurements were made for selected classes in three situations: the size of an object in an array (given by the result of sizeof applied to the class name), the size of an object allocated on the data stack (given by subtracting the addresses of consecutive objects on the stack), and the size of object allocated on the object heap (given by applying sizeof to an object allocated using new). The sizes (in bytes) are:

instance	g++	CC2.1
a in an array	4	4
a on stack	8	4
a on heap	16	16
d15 in an array	64	544
d15 on stack	64	544
d15 on heap	72	552
h in an array	188	1240
h on stack	192	1240
h on heap	200	1248

The following observations were related to the IUE design committee:

- The constructor/destructor calls can slow down by a factor of over a thousand with heavy use of virtual base classes.
- Object size increases dramatically with the number of virtual base classes. The increase is compiler dependent (and might be eliminated by some compiler). An indication of the virtual inheritance size overhead for g++ and CC is given by:
  - g++ overhead = 4 × (number of "virtual" that appear in the class and base class declarations) + (number of base classes that are virtual and add new virtual functions or are not virtual and add the very first virtual function to objects of the base class)
  - (At least quadratic in the number of indirect virtual bases) + 4 × (number of base classes that are not virtual and add the very first virtual function to objects of the base class)

  In contrast to these linear and quadratic space overheads, Lucid Common Lisp has a 24 byte overhead for any CLOS object.

- There is an incremental increase in member data access time for each level of virtual inheritance between the class of variable declaration and class in which the member data is declared. The increment, for all compilers and optimization settings is about 0.12 μsec.
- The presence of virtual base slows method dispatch. Our hierarchies showed worst case slow down by a factor of six (0.6 μsec to 3.6 μsec).
- Virtual base classes complicate the use of classes in C++. There is no simple way to pass constructor arguments to base classes, to downcast, or to traverse all base classes with a fixed operation.

The primary conclusion relevant to IUE design is that for efficiency, the use of tangled inheritance with virtual base classes should be significantly restricted in the IUE.

With Lucid Common Lisp (CLOS) all instances were 24 bytes larger than the number of bytes needed to store the user data. Thus, other than a fixed offset, the increase in instance size with structure of the inheritance hierarchy does not occur. However, with CLOS there is an across-the-board slow down of approximately a factor of 10 with respect to C++. Thus, CLOS can be more storage efficient for large and tangled inheritance hierarchies, but is significantly slower. Also, C++ can be more storage efficient for small or untangled inheritance hierarchies.

We made the following recommendations for structuring the IUE hierarchy (which have helped structure subsequent developments of the C++ IUE):

- **Avoid virtual base classes in subclass of IUE-Object.** This makes instances that are small, that are constructed and destroyed quickly, and execute methods quickly. Further, C++ becomes easier to use: there is a simple way to traverse all base classes, simple down-casting works, and constructor argument passing is simplified. Virtual base classes may still be used for non-IUE-Object superclasses of IUE-Objects. The structural requirement is that the subgraph of subclasses of IUE-Object needs to be a tree. Following this recommendation eliminates some logically correct and useful hierarchies, however the impact on most IU applications will likely be small. Further investigations should be carried out on this structuring issue.
- **Use classes that do not inherit from IUE-Object (i.e., mix-in classes).** Rather than inheriting from IUE-Object, each mix-in class would define its own methods for interacting with the down-casting and recursive equal, I/O, copy, etc. Other classes can be derived from mix-ins through non-virtual (multiple) inheritance. These classes would then have the general capabilities of the IUE-Object without the overhead multiple virtual inheritance. In general, mix-in classes should not have directed instances (they should be abstract classes).

Section 3

Subtyping for Efficient use of C++

The IUE design defines a set of base class capabilities called dynamic attributes. Dynamic attributes are intended to simplify common IU programming tasks and simplify development of the IUE. They are also significant extensions to the C++ language and require extensive support from automatic code generation facilities. We give an overview of the capabilities of dynamic attributes and suggest alternative approaches.

Dynamic attributes have the following goals:

- Implement lists of arbitrary properties for objects, with default values and choices of speed/memory tradeoffs.
- Inherit property value defaults (or find defaults with multi-object keys).
- Transparently implement attributes with slots (member data), with hashtables (no per instance memory for unused attributes), or with function calls.
- Allow getting and setting attribute values and specification of attribute implementation.

The discussion of dynamic attributes in the IUE design document states the following:

- Object-oriented development inevitably leads to the desire to suppress superclass slots and access methods.
- Since suppression of slots and methods is not allowed in object-oriented programming languages, the IUE must address the issue by extending the C++ (and CLOS) object systems.

The dynamic attribute system was originally developed so that "soft" slots (functioning somewhat like C++ member data) could be implemented by global-scope hashtables that take up no space on an individual object. Subclasses would then be free to ignore the slots with no per-instance storage cost. Also, subclasses would be able to redeclare a slot as having inside-the-instance storage (normal C++ data members or "hard" slots) or as having an arbitrary function called upon slot read and write. Further, an extensive initialization procedure was established for supplying default slot values through search across inheritance graphs and across leading subsequences of a sequence of objects.

We believe that the rationale for the necessity of suppressing slots and methods is based on a flawed premise: that a pure mathematical type is correctly modeled by a class that defines storage slots. Instead, a class that defines slots is always specifying *a particular implementation* of an abstract or mathematical type, not the abstract type itself. The design document motivates the need for dynamic attributes with a discussion of two particular Circle and Ellipse classes. The discussion improperly identifies the general concept of a mathematical ellipse with a particular class implemented with two slots for the foci. There is then a clear problem with Circle inheriting from Ellipse, since a Circle needs only a center not two foci. Dynamic attributes are then used to solve this problem by allowing the user to ignore the unneeded slots in a Circle subclass with no wasted space in an instance. The Ellipse class with soft slots would execute methods significantly more slowly due to the hashtable lookup. The new problem with this solution is probably just as great as the problem being solved, but is largely ignored in the motivation. Both problems arise from the initial incorrect modeling step of representing a mathematical ellipse (and, therefore, all its restrict versions) with a class with two specific storage slots.

An alternative approach to Circle/Ellipse problem is to define abstract classes without slots to specify the mathematical properties of objects. Then, concrete classes (classes actually having slots and capable of having usable instances) are defined as subclasses. In this approach, both Circle and Ellipse implementations could inherit from the abstract Ellipse, but the Circle implementation would never directly inherit from the Ellipse implementation with two slots for foci. Figure 3-1 shows one way to organize part of the class hierarchy. The "Acc" classes represent abstract mathematical definitions of Circle and Ellipse. The "Mut" classes represent classes defining the mutator methods that change the slot values of an instance. Circle1 and Ellipse1 represent two (of many possible) classes that can have instances of mutable circles and ellipses. The critical observation is that Circle1 inherits from EllipseAcc, but not from Ellipse1 or from EllipseMut. Therefore, Circle1 inherits all the behavior of an ellipse and nothing extra. Even simpler hierarchies can be constructed if mutation operations are allowed to "fail" when they do not make sense in subclasses. This allows the separate accessor and mutator classes to be merged.

Solutions that use the built-in capabilities of C++ are likely to result in smaller and faster executables. The run-time support for dynamic attributes will likely be quite

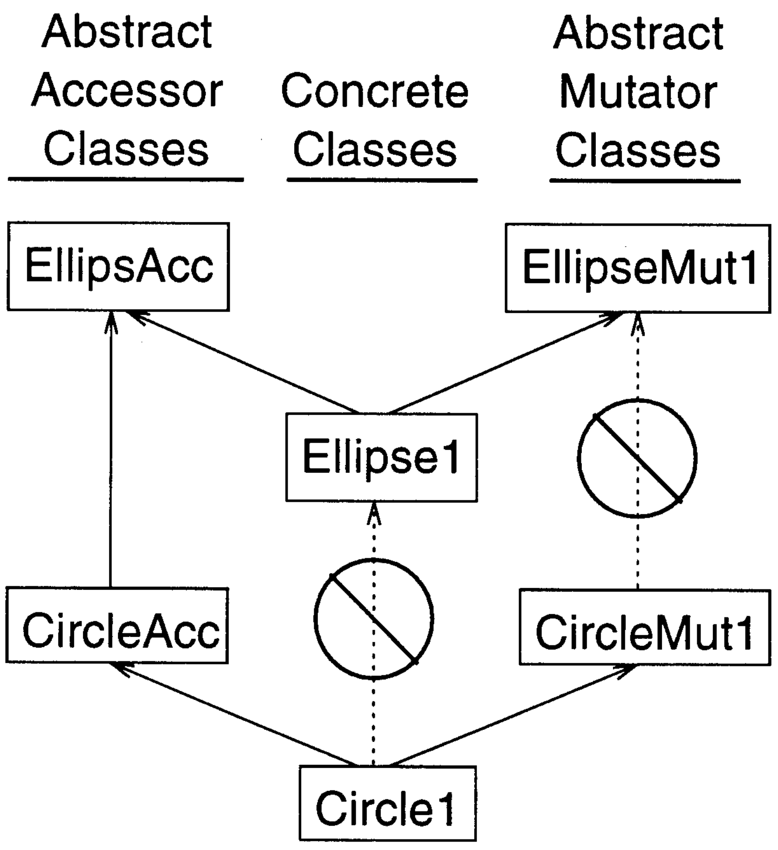


Figure 3-1: Hierarchy modification to handle Circles and Ellipses

complex, making for larger, slower executables and longer compile times. Alternate ways of achieving the aims of dynamic attributes include:

- Separate abstract classes for accessors (read-only) and mutators. Allows "more specialized" objects to have fewer hard slots.
- Develop an explicit class for "soft" slots. Do not attempt to "hide" the implementation of a soft slot from the user, make the user call explicit soft slot methods to store and retrieve data.
- If the initialization strategy of the IUE-context remains a desirable feature, consider implementing it with an IUE-Function with a domain of object sequences. A tree of overriding unions and insertions could approximate IUE-context inheritance and dynamic attribute "copy-on-write."
- Dynamic attributes can store and find values keyed on any leading subsequence of a sequence of objects. Support for this can likely be dropped without affecting users.
- Implement property list objects and make them explicitly available to the user. Do not try to make property lists always look like a regular C++ data member ("hard" slot).

The current IUE design has many places were a developer currently might find he wants to suppress inherited slots and methods. The approach used for the Circle/Ellipse problem can be used to restructure these problem areas so that suppression is not needed. Two examples are:

- The Spatial-Object class defines a bounding-prism slot; however, it may be highly desirable for some subclasses, such as Point, to not waste space on that slot. The solution, without leaving standard C++, is to remove the bounding-prism slot from Spatial-Object and only include the slot in subclasses where it is indeed used. A method that returns a bounding-prism can be included in the basic Spatial-Object.
- Ordered-Pointset defines a pts slot for a sequence; however, many subclasses may not need a sequence (e.g., an interval might be best defined by formulas rather than an explicit sequence). A new sequence might be better generated whenever needed.

Inheritance structures could eliminate the primary problem addressed by dynamic attributes without going beyond the capabilities of standard C++. A combination of disciplined design and development of useful classes that the user can use in standard C++ ways should go far in eliminating any need for the dynamic attribute mechanism.

Section 4

Liaison to the Open Object Oriented Database Project

BAH acted as the liaison between the ARPA/Texas Instruments Open Object Oriented Database (Open OODB) project activities and the IUE design committee. BAH personnel attended Open OODB design workshops and reviewed Open OODB documentation for the IUE. This section describes a statement of IUE needs transmitted from BAH to the Open OODB design workshop.

The IUE will be selecting an OODB for integration into many system developments. Rather than developing an OODB, we prefer to use OODB standards and their commercial implementations. For the IUE, development would have high cost, high technical risk, and low likelihood of widespread acceptance. Unfortunately, current OODBs meet few of the IUE functional and structural requirements. These requirements include C++ and Lisp compatibility, an underlying OO data model, and ability to customize the database for efficient storage and retrieval into any underlying storage medium, including a Relational Database Management System (RDBMS).

The ARPA/TI Open OODB development looks promising. The strawman Open OODB addresses concerns similar to those of the IUE, and the collaborative design process allows the IUE to contribute to requirements definition, design, and implementation. Further, the active participation of many organizations will greatly enhance general acceptance of the Open OODB standards and implementations.

The next section reviews requirements on an OODB to support IU. The section following that discusses the need and suggested approach for abstract class specifications for an OODB.

4.1 Requirements for Image Understanding

IU stresses Object Oriented (OO) concepts and implementations. IU uses models of the real world to generate and evaluate hypotheses that match model to sensor-obtained data. Objects are created by feature extraction and by complex reasoning processes involving numerous other objects. One IU operation can easily generate tens of Mbytes of intermediate and result objects. IU classes are varied, including:

- Hierarchical spatial objects with many attributes
- Networks (adjacency, Bayesian, relational, coordinate systems)
- User interface objects
- Images
- Primitive and composite extracted features (edges, curves, regions, etc.)
- Hypotheses.

Efficiency is an overriding concern with IU systems. Potential efficiency problems that an OODB must address include:

- Objects with volumes of redundant data cached for computational efficiency
- Large quantities of objects (typically "small") that are used for their value only and are not shared by other objects.

An object can reference volumes of redundant data. For example, a cylinder object defined by six real numbers might reference 1,000 approximating triangular facets. If other objects do not rely on maintaining identity of facets, the facets need not be stored. Facets can generally be recomputed much faster than they can be moved to and from disks (especially as persistent objects). Further, storing the facets wastes disk space. An OODB for IU must provide hooks (in the form of generic or virtual functions) for controlling the transformation of objects to and from equivalent storage forms.

IU generates many objects, especially small built-in primitives, such as cons cells or structs. If an object's identity is to be shared among multiple objects (either for equality testing or shared interface), that object has to be made persistent. However, applications often specifically disallow sharing of certain objects and these can then be stored by value. This allows for immense savings in symbol table size and access time, I/O translation (formatting) time, and disk storage. An OODB for IU must provide hooks for specifying which objects have to be persistent and which can be stored by "value."

IU uses network maintenance or view access/update capabilities. This involves objects defined by operations on other objects in which the semantics call for the values of all related objects to change together. For efficiency, IU applications cache dependent value information. Then, changes in underlying object values must result cache update or notification of cache invalidation. These capabilities are often used in networks and in user interface views. Read-only views have been sufficient for all but the user interface portions of our IU applications.

Because IU systems generate many objects connected in many different ways, an OODB should support object removal, garbage collection, and integrity checks.

Since IU is a rapidly changing research area with ongoing research on object representations, an OODB should support updating class definitions (schema evolution).

An OODB for IU (particularly for Test and Evaluation) should be powerful enough to implement an RDBMS. This should be done as an integrated collection of classes for domain, attribute, tuple, table, dependency, constraint, index, database, etc. To capture the semantics of an RDBMS, the system must support storage of object identity (e.g., for databases, tables, and attributes) and object value without identity (e.g., for tuples).

An OODB for IU must support efficient disk I/O for large contiguous-memory data objects such as image arrays.

4.2 Class Specification for an OODB

A primary requirement for an OODB is that the semantics of all objects be unaffected by storage actions of the OODB. For example, an object must mean the same thing before and after a commit. Since an efficient OODB can change object representation, ignore redundant data, and even ignore object identity, miscommunication between the OODB translator and the class design can result in the OODB cutting too many corners and fouling the class design. Conversely, cutting too few corners can devastate efficiency.

We suggest that a consistent OODB requires an abstract specification of the semantics of the object classes (i.e., a "data model" per class expressed in a specification language). A class specification implies just what procedural accesses and OODB translation short cuts are allowed. If data is to be shared between different applications, perhaps in different languages, it is necessary that the specification be independent of programming language: neither C++ nor CLOS can be "the data model." It is not necessary that specification be automatically transformed into code or storage translators.

Specification is a foundation of RDBMSs and a primary reason for their widespread use. The relational data model is expressive enough for many applications and there is no doubt about the meaning of the database before and after an RDBMS operation.

OO classes need similar specifications to support safe and efficient OODBs. Although the theory of specification is well developed for systems without inheritance, there is not yet general agreement on how to write OO class specifications. We propose that the approach of viewing a relational table as a time-varying object that at any moment refers to an algebraically well-defined abstract relation extends to more general classes. That is, an OO class defines time-varying objects that refer to algebraically defined abstract objects. Further, inheritance allows new classes to be developed that modify either the underlying algebra, the mapping from OO objects to abstract objects, or the implementing code.

The programming language Eiffel follows a similar, but less formal, approach using preconditions, postconditions, and class and representation invariants. The combination of CLU and Larch provide similar facilities, but without inheritance. C++ and CLOS enforce virtually no semantic constraints on their inheritance hierarchies. This makes them inherently dangerous in OODB environments and increases the importance of clear, external specifications.

The issues that should be addressed with specifications include:

- Specification of an Abstract Data Type (ADT) (i.e., immutable abstract objects with sorts, operations, operation interfaces/templates, and axioms)
- Specification of a class (in terms of function-varying links to ADTs)
- Specification of functions specializing on multiple arguments
- Distinct things inheritance augments (ADT axioms, ADT interface and axioms, class implementation, class interface, and class axioms)
- Object identity as an ADT
- Mutable and immutable objects
- Hidden interfaces, object sharing, views, and truth maintenance
- Context dependent definitions of equal?, mutable?, copy, read, and write
- Efficient and consistent multiple representations.

Section 5

Prototyping of Base Class in CLOS

BAH prototyped a set of base classes in CommonLISP. The classes included iue-object, set, finite-set, boolean, ident-set, ident-hash-set, relation, and iue-function. The implementation brought to light several problems with the original definitions of base classes. Solutions were developed and incorporated in the implementation. These included:

- Making both object-mutating and new-instance-creating versions of functions (appended "-NEW" for new-instance-creating versions).
- Adding a function that would map other functions over IUE-sets.
- Adding Generic make-<class-name> construction methods (e.g., MAKE-IDENT-HASH-SET).
- Adding subclasses of iue-set: finite-set, ident-set, ident-hash-set
- Adding a boolean class.
- Resolving name conflicts with CommonLISP. Conflicting names included AND, DESCRIBE, GET, INSPECT, INTERSECTION, MAP, MAPC, REMOVE, SELECT, NOT, OR, UNION, WRITE, and XOR. The conflict was resolved by appended "G" (for Generic) to iue versions.
- Grouping export's, defclass's, defgeneric's, defmethod's.
    - Files named "<class-name>-def.lisp" contain one defclass, new defgenerics for the class, plus new exports
    - Files named "<class-name>.lisp" file would contained new defmethods.
    - Further effort should be taken to segregate and order the loading of: exports, imports, macros, metaclasses, classes, methods, defgenerics, and globals.

Table 5-1 lists the classes and the methods which were implemented in CLOS.

Table 5-1: Classes and methods prototyped in CLOS

class	methods
boolean	DEEP-COPY
	SHALLOW-COPY
	DEEPEN-SHALLOW-COPY
	COPY
	COPY-LIKE
	IS-EQUAL
	IS-UNIQUE
	IS-IMMUTABLE
	CHANGE-TO-IMMUTABLE
	IS-LOCKED
	SET-LOCK
	NOT-IUE
	AND-IUE
	XOR-IUE
	OR-IUE
FINITE-SET	IS-EQUAL
	IS-FINITE
	IS-COUNTABLE
	IS-COUNTABLY-INFINITE
	IS-UNCOUNTABLE

Table 5-1 (Continued): Classes and methods prototyped in CLOS.

class	methods
IDENT-HASH-SET	MAKE-INSTANCE-2
	MAKE-IDENT-HASH-SET
	MAKE-INSTANCE-2
	MAKE-IDENT-HASH-SET
	MAKE-INSTANCE-2
	MAKE-IDENT-HASH-SET
	MAKE-INSTANCE-2
	MAKE-IDENT-HASH-SET
	COPY
	COPY-LIKE
	CARD
	IN
	DISJOINT
	IS-SUBSET
	INSERT
	REMOVE-IUE
	UNION-NEW
	UNION-IUE
	DIFFERENCE-NEW
	DIFFERENCE
	INTERSECTION-NEW
	INTERSECTION-IUE
	SYMMETRIC-DIFFERENCE-NEW
	SYMMETRIC-DIFFERENCE
	MAPC-IUE
	MAPC-IUE

Table 5-1 (Continued): Classes and methods prototyped in CLOS.

class	methods
IDENT-SET	EQUIVALENT-ARGS UNION-COMPATIBLE INSERT-COMPATIBLE IS-SINGLETON IS-EMPTY
IUE-CLASS	IN MAKE-INSTANCE-2
IUE-FUNCTION	
IUE-OBJECT	IS-SAME IS-VALID SET-VALID IS-KIND-OF OF-CLASS OF-SUBCLASS IS-UNIQUE IS-IMMUTABLE CHANGE-TO-IMMUTABLE IS-LOCKED SET-LOCK DESTROY
RELATION	
SET	

Section 6

Mathematics-Oriented Base Class Designs

BAH refined the mathematical foundation for base class designs beyond what has been agreed upon for inclusion in the IUE Class Definition and IUE Overview documents. This section presents an overview of the designs for Sets, Relations, Functions, probability-related classes, and Manifolds (possible replacements for Spatial-Objects).

The Set classes were refined with goals of removing circular definitions involved in previous versions and of making all methods on a Set or Set subclass depend strictly on information derivable from just the elements of the set. Achieving this latter goal ensures that Sets do not depend on non-set-like associated data or slots. The intent is to define Set and its subclasses so that all the important auxiliary sets (such as range of a function) could, in principle, be derived from the primary object's state, or set of values. For instance, the exact range of a function can be extracted from a function by repeatedly evaluating the function. This property makes it unnecessary to augment the natural state of a set-related object by defining slots to hold auxiliary information. An implementation could still define such slots to improve efficiency, but the slot would hold essentially redundant information.

Probabilistic quantities and Bayesian network classes were defined. The defined classes include random variables, conditional distributions, conditional densities, and Bayesian networks. The current IUE design document does not include definitions for these objects. The definitions given here give the programmer a simple interface for stating most interesting operations on Bayesian networks.

The manifold class generalizes spatial objects by representing any continuously parameterizable set. The definition includes definitions of coordinate systems, sets of compatible coordinate systems, and linearized versions of both spaces and transformations.

6.1 Sets, Functions, and Relations

This section describes the IUE objects derived from elementary set and relation theory. The objects described include: set, function, scheme, tuple, relation, and Cartesian product. Later sections describe objects closer to the domain of IU and probabilistic inference.

High-level inspection operations that yield true or false and take any type of object as arguments include:

- Equality test: *Is-Equal*$(\cdot, \cdot)$.
- Type tests: *Is-Set*$(\cdot)$, *Is-Function*$(\cdot)$, *Is-Scheme*$(\cdot)$, *Is-Tuple*$(\cdot)$, *Is-Ordered-Pair*$(\cdot)$, *Is-Relation*$(\cdot)$, *Is-Cartesian-Product*$(\cdot)$, *Is-Random-Variable*$(\cdot)$, *Is-Conditional-Distribution*$(\cdot)$, *Is-Discrete-Conditional-Distribution*$(\cdot)$, *Is-Gaussian-Distribution*$(\cdot)$, *Is-Linear-Gaussian-Conditional-Distribution*$(\cdot)$, *Is-Multi-modal-Linear-Gaussian-Conditional-Distribution*$(\cdot)$, *Is-Manifold*$(\cdot)$, *Is-Chart*$(\cdot)$, etc.

A **set** represents an unordered collection of distinct elements. Common denotations for a set are $\{ x \mid \phi(x) \}$ and $\{ f(x) \mid \phi(x) \}$ for a predicate ϕ and function f. Operations taking sets as arguments include:

- Boolean combinations: *Union*(A, B), *Intersect*(A, B), *Difference*(A, B), *Symmetric-Difference*(A, B) (i.e., $\cup$, $\cap$, $\setminus$, and Δ).
- Union and intersect on sets of sets: *Union-Set*$(\mathcal{A})$ and *Intersect-Set*$(\mathcal{A})$ (i.e., $\bigcup_{A \in \mathcal{A}} A$ and $\bigcap_{A \in \mathcal{A}} A$).
- Subset test: *Is-Subset*(A, B), returning true if $A \subset B$.
- Intersection test: *Is-Disjoint*(A, B), returning true if $A \cap B = \emptyset$.
- Membership test: *In*(A, x) (i.e., "Is $x \in A$?").
- Cardinality: *Card*$(A) \equiv |A|$.
- Cardinality tests: *Is-Singleton*(A), *Is-Empty*(A), *Is-Finite*(A), *Is-Countable*(A), *Is-Countably-Infinite*(A), *Is-Uncountable*(A).
- Subset selection: *Select*$(A,$ *Boolean-Test*$(\cdot)) \equiv \{ a \mid a \in A$ and *Boolean-Test*$(a) =$ true $\})$.
- Empty set: *Empty-Set*$() \equiv$ the empty set $\equiv \emptyset$.

- Power set: $\mathit{Power\text{-}Set}(A) \equiv \mathcal{P}(A)$, the set of all subsets of A.
- Identity function: $\mathit{Identity\text{-}Function}(A)$ yields the function (function is defined below) $f: A \rightarrow A$ satisfying $f(a) = a$ for all $a \in A$.
- Constant function: $\mathit{Constant\text{-}Function}(A, b)$ yields the function $f: A \rightarrow \{b\}$ satisfying $f(a) = b$ for all $a \in A$.
- Set-to-Relation: $\mathit{Set\text{-}To\text{-}Relation}(A, x) \equiv x{:}\,A \equiv \{\,(x, a) \mid a \in A\,\}$. This operation maps a set A and object x into a relation. If R is the resulting relation, it has x as its only attribute and it satisfies $\mathit{Relation\text{-}Dom}(R, x) = A$. See below for definitions of relation, attribute, and *Relation-Dom*.
- Operations to create all functions from A to B, all 1-1 functions, all onto functions, and all invertible functions.
- Choose: $\mathit{Choose}(A) \equiv$ an arbitrary element of A.

Uses of sets include: sets of site objects, sets of images, sets of features, sets of random variables, sets of conditional distributions, range and domain sets of functions, and sets of tuples in relations, sample spaces, and functions.

A **function** represents a functional (i.e., many-to-one) mapping from a domain set to a range set. A function establishes a pairing that maps each element of its domain to exactly one element of its range. The notation $f{:}A \rightarrow B$ is equivalent to the three conditions that f is a function, A is the exact domain of definition of f, and B contains the exact range of f. Such an f is also said to be a B-valued function on A. Further, $A \rightarrow B$ denotes the set of all functions mapping A to B. If $f: A \rightarrow B$ and $a \in A$, then the unique element of B associated with a is written as $f(a)$ or fa. Note, we will use a centered dot (e.g., $f \cdot g$) for multiplications having a function on the left.

Every function is uniquely associated with the set of ordered pairs (ordered pairs are defined below) called its graph:

$$f \leftrightarrow \{\,(a, f(a)) \mid a \text{ is in the domain of } f\,\}.$$

The graph is a type of relation (relation is defined below as a type of set with tuple elements). It is convenient to elevate the status of this particular association between graph and function and say that a function "is" its graph. This identification of function with relation eliminates explicit conversions between function and associated relation and allows operations defined for relations (or sets) to apply directly to functions. To avoid circular definitions, the definition of relation given below depends on the concepts of domain, range, and mapping property of functions, but not on modeling a function as a graph, relation, or set.

Operations taking functions (with $f: A \rightarrow B$ and $g: C \rightarrow D$ representing general functions) as arguments include:

- *Evaluate*$(f,a) \equiv f(a) \equiv fa \equiv$ the unique element of B associated with $a \in A$.
- *Evaluate-Set*$(f,C) \equiv$ image of set $C \subset A$ under $f \equiv \{\, f(a) \mid a \in C \,\}$ where $C \subset A$. With sufficient context, this operation may also be written as $f(C)$.
- *Point-Set-Invert*$(f) \equiv$ the function mapping elements of *Ran*(f) into inverse images (subsets of A) $\equiv h$ such that h: *Ran*$(f) \rightarrow$ *Power-Set*(A) and $a \in h(b)$ if and only if $f(a) = b$.
- *Set-Set-Invert*$(f) \equiv$ the function mapping subsets of *Ran*(f) into inverse images (subsets of A) $\equiv h$ such that h: *Power-Set*(*Ran*(f)) $\rightarrow$ *Power-Set*(A) and $a \in h(C)$ if and only if $f(a) \in C$ and $C \subset$ *Ran*(f).
- *Compose*$(g,f) \equiv g \circ f \equiv$ function with domain $A \cap f^{-1}(B \cap C)$ that evaluates to $g(f(a))$ for all a in the domain.
- *Dom*$(f) \equiv A \equiv$ the (exact) domain of $f \equiv$ *Evaluate-Set*(*First*$,f$). *First*$(\cdot)$ is a function on ordered pairs and is defined below.
- *Ran*$(f) \equiv$ the (exact) range of $f \equiv \{\, b \mid a \in$ *Dom*(f) and $f(a) = b \,\}$ $\equiv$ *Evaluate-Set*$(f,$ *Dom*$(f))$ $\equiv$ *Evaluate-Set*(*Second*$,f$). B always contains *Ran*(f). *Second*$(\cdot)$ is a function on ordered pairs and is defined below.
- *Domain*$(f) \equiv$ some superset of *Dom*(f).
- *Range*$(f) \equiv$ *Ran*$(f) \equiv$ some set containing *Ran*(f) (usually B).
- *Restrict*$(f,D) \equiv$ restriction of f to the domain $A \cap D \equiv h$: $(A \cap D) \rightarrow B$ such that $h(d) = f(d)$ for $d \in A \cap D \equiv$ *Select*(f,e) where $e((x,y))$ is true if $x \in A \cap D$.
- *Overriding-Union*$(f,g) \equiv f \cup$ *Restrict*$(g,$ *Dom*$(g) \backslash$ *Dom*$(f))$.
- Tests for function type: *Is-1-To-1*, *Is-Onto*, and *Is-Invertible*.

The values of *Dom*(f) and *Ran*(f) are uniquely determined by the function, f. However, the values of *Domain*(f) and *Range*(f) may depend on both f and the context in which the expressions appear. "The" domain or range of f refers to *Dom*(f) or *Ran*(f) and "a" domain or range of f refers to *Domain*(f) or *Range*(f).

A function f is one-to-one when every element of *Ran*(*Point-Set-Invert*(f)) is a singleton set. For one-to-one functions, we define the operation *Invert* as:

- *Invert*$(f) \equiv$ the function mapping elements of *Ran*(f) into unique inverse elements of $A \equiv f^{-1}$ such that f^{-1}: *Ran*$(f) \rightarrow A$ and $f^{-1}f(a) = a$ for all $a \in A$.

Note that every function f is onto $Ran(f)$, but not necessary onto $Range(f)$. $Invert(f)$ has $Ran(f)$ as its exact domain.

Many functions of interest have range sets containing functions sharing one domain (e.g., in $f: A \rightarrow (B \rightarrow C)$, B is the unique domain of all functions in the range of f). Examples of functions with such ranges of sets of functions include random variable-valued functions (see Section 6.2) and images with an operation kernel per pixel [Ritter 1990]. Let X be a set of functions all having domain B. If $C = \bigcup_{x \in X} Dom(x)$, then we have $x: B \rightarrow C$ for all $x \in X$. Let f be a function-valued function such that $f: A \rightarrow X$, or equivalently $f: A \rightarrow (B \rightarrow C)$. We use a $'$ to denote the operation that produces an associated function-valued function $f': B \rightarrow (A \rightarrow C)$ satisfying

$$f'(b)(a) = f(a)(b) \text{ for all } b \in B \text{ and } a \in A.$$

Two well-motivated names for the $'$ operation are dispatch and transpose. Dispatch ([Meyer 1990] pg. 35) refers to the act of "dispatching" an argument (e.g., b) through all functions in the range to build a function of results. Transpose ([Ritter 1990]) might refer either to the transposition of arguments between f and f' or to an analogy to matrix transpose. To avoid confusion with the common usage of transpose dealing with operators on dual spaces, we use the name *Dispatch* for the operation. *Dispatch* is self inverting and satisfies:

- $Dispatch(f) \equiv f'$ and $f'' = f$.

Given an operation on a given domain D, it is common practice to extend the operation to functions returning elements of D. For example, "+" is originally defined for real numbers, but is then extended to apply to any combination of real numbers and real-valued functions. Such extended (or induced) operations are given as follows. We define *Induce* such that if $f: A \rightarrow B$ and $\mathcal{C}$ is a set of sets of interest, then

$$\begin{aligned} Induce \ : \ & A \cup (\bigcup_{C_0 \in \mathcal{C}} (C_0 \rightarrow A)) \cup (\bigcup_{C_0, C_1 \in \mathcal{C}} (C_1 \rightarrow (C_0 \rightarrow A)) \cup \cdots \\ & \rightarrow B \cup (\bigcup_{C_0 \in \mathcal{C}} (C_0 \rightarrow B)) \cup (\bigcup_{C_0, C_1 \in \mathcal{C}} (C_1 \rightarrow (C_0 \rightarrow B)) \cup \cdots \end{aligned}$$

such that if $g: (C_n \rightarrow \cdots (C_1 \rightarrow (C_0 \rightarrow A) \cdots)$, then

$$Induce(f)(g): (C_n \rightarrow \cdots (C_1 \rightarrow (C_0 \rightarrow B) \cdots)$$

such that for all $c_i \in C_i$,

$$Induce(f)(g)(c_n)(c_{n-1}) \cdots (c_0) = f((g)(c_n)(c_{n-1}) \cdots (c_0)).$$

Induce can also be extended to multi-argument functions, such as "+".

A **scheme** is a function (usually with finite domain) for which all elements in the range are sets. The elements of the domain of a scheme are called attributes (of the scheme). If A is an attribute of a scheme S, then the set $S(A)$ is called the domain of A (with respect to S). Operations taking schemes as arguments include:

- *Cartesian-Product*$(S) \equiv$ the Cartesian product having relation scheme S (Cartesian product and relation scheme are defined below).

For a finite scheme $S = \{\, (a_i, D_i) \mid 0 \leq i \leq n \,\}$, the Cartesian product (certain sets of tuples defined in detail below) with scheme S may be written as $(a_0\colon D_0) \bowtie (a_1\colon D_1) \bowtie \cdots \bowtie (a_n\colon D_n) = \bowtie_{i=0}^{n-1} a_i\colon D_0$, where the join operation for relations ($\bowtie$) is defined below.

A **tuple** is a function (usually with finite domain) associated (if only by context) with a scheme. A scheme S is a tuple scheme of tuple T if and only if

$$Dom(T) = Dom(S) \text{ and } T(A) \in S(A) \text{ for all } A \in Dom(S)$$

or, equivalently, if and only if

$$T \in \textit{Cartesian-Product}(S).$$

Operations taking tuples as arguments (beyond those operations already defined that take functions as arguments) include:

- *Has-Tuple-Scheme*$(T, S) \equiv$ test for scheme S being a tuple scheme of tuple T.
- *Tuple-Scheme*$(T) \equiv$ some tuple scheme of T with specific choice depending on context.

An **ordered pair** is a tuple with domain $\{0, 1\}$. The Cartesian product of ordered pairs, $0\colon A_0 \bowtie 1\colon A_1$, may be abbreviated to $A_0 \times A_1$. If c is an ordered pair, $c(0) = a$, and $c(1) = b$, then we write $c = (a, b)$. Operations that take ordered pairs as arguments include:

- *First*$(c) \equiv c(0) \equiv a$.
- *Second*$(c) \equiv c(1) \equiv b$.

Note: if a Common Lisp cons cells were used to represent an ordered pair, the operation *First* would be represented by the Common Lisp function `first`, but *Second* would be represented by `rest`.

An **ordered tuple** is a tuple with exact domain $\{\, i \mid 0 \leq i \leq n-1 \,\}$ for some $n \geq 1$. An ordered n-tuple, c, can be written as $c = (c(0), c(1), \ldots, c(n-1))$. The notation for a Cartesian product of ordered tuples can be abbreviated to $D_0 \times D_1 \times \cdots \times D_{n-1} = \times_{i=0}^{n-1} D_i$.

A **relation** is a set of tuples with all tuples sharing a common tuple scheme. If the scheme S is a tuple scheme of all tuples in a relation R, we say that S is a relation scheme of R. Let R and R_1 be relations with exact relation schemes S and S_1 and $a \in \mathit{Dom}(S)$. Operations that take relations as arguments include:

- $\mathit{Relation\text{-}Sch}(R) \equiv S \equiv$ the (exact) relation scheme of R.
- $\mathit{Relation\text{-}Scheme}(R) \equiv$ some relation scheme of R.
- $\mathit{Relation\text{-}Attributes}(R) \equiv \mathit{Dom}(S) \equiv$ relation attributes of R.
- $\mathit{Relation\text{-}Dom}(R, a) \equiv \mathit{Relation\text{-}Sch}(R)(a)$.
- $\mathit{Relation\text{-}Domain}(R, a) \equiv \mathit{Relation\text{-}Scheme}(R)(a)$.
- $\mathit{Join}(R, R_1) \equiv R \bowtie R_1 \equiv$ relational natural join of R and $R_1 \equiv \{\, \tau \cup \tau_1 \mid \tau \in R,\ \tau_1 \in R_1,$ and $\tau(a) = \tau_1(a)$ for all $a \in \mathit{Dom}(S) \cap \mathit{Dom}(S_1) \,\}$.
- $\mathit{Relation\text{-}Project\text{-}On}(R, A) \equiv$ relational projection onto the attributes of $A \cap \mathit{Relation\text{-}Attributes}(R) \equiv \{\{\ (a, d) \in \tau \mid a \in \mathit{Dom}(S) \cap A \,\} \mid \tau \in R \,\}$.
- $\mathit{Relation\text{-}Project\text{-}Off}(R, A) \equiv$ relational projection onto $\mathit{Relation\text{-}Attributes}(R) \backslash A \equiv \{\{\ (a, d) \in \tau \mid a \in \mathit{Dom}(S) \backslash A \,\} \mid \tau \in R \,\}$.
- $\mathit{Rename\text{-}Attributes}(R, f) \equiv \{\, (f(a), d) \mid (a, d) \in R \,\}$ where f is a 1-1 function with domain containing the relation attributes of R.

The value of $\mathit{Relation\text{-}Sch}(R)$ is uniquely determined by R. The value of $\mathit{Relation\text{-}Scheme}(R)$ may depend on both R and context. Also, note that the relation domain denotes an element found in the range (not domain) of the relation scheme.

A relation R with single attribute x and $\mathit{Relation\text{-}Dom}(R, x) = D$ is written as $R = x\colon D$.

A functional dependency (FD) of a relation is a pair of subsets of relation attributes such that the values of the first subset of attributes functionally determine the values of the second subset. That is, the pair (A, B) is a functional dependency of a relation R if and only if A and B are both subsets of $\mathit{Relation\text{-}Attributes}(R)$ and for all τ_1 and τ_2 with equal restrictions to A, τ_1 and τ_2 also have equal restrictions to $A \cup B$. If (A, B) is a functional dependency of R, then A and B are called the sets of independent and dependent attributes (with respect to (A, B)). A relation R with functional dependency (A, B) can be an argument to operations that include:

- $\mathit{Evaluate\text{-}FD}(R, (A, B), a) \equiv$ the tuple with attributes B corresponding to tuples in R with the tuple a corresponding to the value of attributes A.

- *Has-FD*$(R, (A, B)) \equiv$ the test for whether (A, B) is a functional dependency of R.

A **Cartesian product** the "largest" relation for a given relation scheme. That is, for a scheme S, *Cartesian-Product*(S) is the union of all relations with relation scheme S. We also have:

$$\textit{Cartesian-Product}(S) = \{\, (a, d) \mid a \in \textit{Dom}(S) \text{ and } d \in S(a) \,\}$$

and if $S = \{\, (a_i, D_i) \mid i = 0, \ldots, n-1 \,\}$, then

$$\textit{Cartesian-Product}(S) = \bowtie_{i=0}^{n-1} a_i{:}\, D_i.$$

The **reals** is the set $\mathbb{R}$ of real numbers. If the scheme S is given by $S{:}\, \{0, \ldots, n-1\} \to \mathbb{R}$, then the n**-dimensional reals** is given by *Cartesian-Product*(S). Notice that $\mathbb{R}^1$ is not equal to $\mathbb{R}$: $\mathbb{R}^1 = 0{:}\,\mathbb{R}$ is a relation and $\mathbb{R} = \textit{Relation-Dom}(\mathbb{R}^1)$ is a simple set. The reals and n-dimensional reals are returned by:

- *Reals*$() \equiv \mathbb{R}$.
- *Reals-N*$(n) \equiv \mathbb{R}^n$.

Other sets of extended reals and integers such as $[a, b]$, $[a, b)$, $(a, b]$, (a, b), $i{:}\,j$, and $\mathbb{R}^+$ with an object for ∞ should be defined.

6.2 Probability Spaces and Random Variables

Probability theory is structured around random variables defined on sample spaces that have associated probability measures and σ-algebras of measurable sets. This section reviews the relevant facts: the detailed mathematics of the theory can be found in references such as [Loève 1977].

A **probability space** is a combination of a set (called the **sample space**), a σ-algebra on the set, and a probability measure on the σ-algebra. With our notion of function, a probability measure is equivalent to a probability space (the measure's domain yields the σ-algebra and a *Union-Set* on the σ-algebra yields to sample space). A sample space may be a Cartesian product with relation attributes corresponding to phenomena that span (perhaps redundantly) all degrees of freedom of stochastic variation in a situation of interest.

A **random variable** (or **random relation** with respect to a probability space (probability measure) is a relation such that 1) a functional dependency exists that has one independent attribute known as Sample-Space, 2) the domain of Sample-Space is the sample space, 3) the dependent attributes are all remaining relation attributes, and 4) the function associated with the functional dependency is measurable. Each dependent attribute, also called a random attribute, typically names a stochastic phenomenon of interest (e.g., length of a box of uncertain size). Although a random variable has the Sample-Space independent attribute, the *Evaluate-FD* operation is rarely required (or even computable). Operations of more consistent interest for probabilistic analysis are *Relation-Dom* and *Relation-Domain* of the random attributes.

In applying relational operations to random variables, the Sample-Space attribute almost always appears in a fixed pattern, making it almost superfluous. To simplify the expressions, we define several operations as slight modifications of existing operations. The operations that take a random variable (X) as an argument include:

- $\textit{RV-Ran}(X) \equiv \textit{Relation-Project-Off}(X, \{\textit{Sample-Space}\})$.
- $\textit{RV-Range}(X) \equiv$ some relation containing $\textit{RV-Ran}(X)$. Usually, this operation will return a Cartesian product.
- $\textit{RV-Sch}(X) \equiv \textit{Relation-Sch}(\textit{RV-Ran}(X))$.
- $\textit{RV-Scheme}(X) \equiv \textit{Relation-Scheme}(\textit{RV-Ran}(X))$.
- $\textit{RV-Attributes}(X) \equiv \textit{Relation-Attributes}(\textit{RV-Ran}(X))$.
- $\textit{RV-Dom}(X, a) \equiv \textit{Relation-Dom}(\textit{RV-Ran}(X), a)$.
- $\textit{RV-Domain}(X, a) \equiv \textit{Relation-Domain}(\textit{RV-Ran}(X), a)$.
- $\textit{RV-Project-On}(X, A) \equiv \textit{Relation-Project-On}(X, A \cup \{\textit{Sample-Space}\})$.
- $\textit{RV-Project-Off}(X, A) \equiv \textit{Relation-Project-Off}(X, A)$.
- $\textit{RV-Difference}(X, Y) \equiv X \backslash\backslash Y \equiv$
  $\textit{RV-Project-On}(X \bowtie Y, \textit{RV-Attributes}(X) \backslash \textit{RV-Attributes}(Y))$.

The standard relational join is generally the correct operation for combining two random variables.

Two random variables, X and Y, are said to be disjoint if $\textit{RV-Attributes}(X) \cap \textit{RV-Attributes}(Y) = \emptyset$. A random variable with each *RV-Dom* being $\mathbb{R}$ and the measure being absolutely continuous with respect to Lebesgue measure, is called a continuous random variable. If each *RV-Dom* is a finite set, the random variable is called discrete. With a mixture of the above two conditions, the variable is called mixed.

6.3 Conditional Distributions

Conditional distributions reflect the "probabilistic" behavior of random variables. This section begins by describing the structure and operations common to all conditional distributions. Subsections then define several specific conditional distributions. The subsections cover:

- **Gaussian.** This section describes the conditional distribution for jointly Gaussian random variables.
- **Linear Gaussian.** This section describes the conditional distribution of random variables that are linear combinations of conditioning random variables and independent joint Gaussian random variables.
- **Discrete.** This sections describes the conditional distribution of finite discrete random variables.
- **Multi-modal Linear Gaussian.** This section extends the linear Gaussian analysis to the case of the parameters of the linear combinations depending on discrete random variables.

If X and Y are random variables and $\mathcal{F}_X$ is the σ-algebra of the measurable subsets containing $\mathit{RV\text{-}Ran}(X)$, then the **conditional distribution** $P(\,X \mid Y\,)$ is, $P(\,X \mid Y\,)$ is the parameterized probability measure:

$$P(\,X \mid Y\,)\colon \mathcal{F}_X \times \mathit{RV\text{-}Ran}(Y) \rightarrow [0, 1]$$

such that for integrable functions $f\colon \mathit{RV\text{-}Ran}(X) \times \mathit{RV\text{-}Ran}(Y) \rightarrow \mathbb{R}$ we have:

$$\int P(Y)(dy) \int P(\,X \mid Y\,)(dx) f(x,y) = \int P(X,Y)(d(x,y)) f(x,y).$$

X is called the conditioned random variable and Y is called the conditioning random variable. Expressions such as $P(\,X_1, X_2 \mid Y_1, Y_2\,)$ mean $P(\,X_1 \bowtie X_2 \mid Y_1 \bowtie Y_2\,)$. We also introduce the notation:

$$P(\,X \in A \mid Y = y\,) \equiv P(\,X \mid Y\,)(A, y).$$

If $P(\,X \mid Y\,)$ is absolutely continuous with respect to a **parameterized measure** $\mu(\,X \mid Y\,)$ with

$$\mu(\,X \mid Y\,)\colon \mathcal{F}_X \times \mathit{RV\text{-}Ran}(Y) \rightarrow [0, \infty],$$

then there is a **conditional density function** $p(\,X \mid Y\,)$ with respect μ associated with $P(\,X \mid Y\,)$

$$p(\,X \mid Y\,)\colon \mathit{RV\text{-}Ran}(X) \times \mathit{RV\text{-}Ran}(Y) \rightarrow [0, \infty]$$

such that

$$\int P(X \mid Y = y)(dx)f(x) = \int \mu(X \mid Y = y)(dx)p(X = x \mid Y = y)f(x).$$

Bayesian inference can be expressed in terms of a small number of operations applied to conditional distributions. These operations are:

- Bayesian Decompose (with Marginal and Conditional special cases)
- Bayesian Compose
- Substitute (with Observe special case)

In the following assume that X, Y, and Z are disjoint random variables and that integrands are such that all integrals exist. The operations that take conditional distributions as arguments include:

- *Vars*(). The set of random attributes of the conditioned random variable is obtained as: $\mathit{Vars}(P(X \mid Y)) = \mathit{RV\text{-}Attributes}(X)$. The set can be obtained by inspection from the domain of $P(X \mid Y)$.

- *Conds*(). The set of random attributes of the conditioning random variable is obtained as: $\mathit{Conds}(P(X \mid Y)) = \mathit{RV\text{-}Attributes}(Y)$. The set can be obtained by inspection from the domain of $P(X \mid Y)$.

- Bayesian Decompose. Let $\mathcal{Y} = \{y_i \mid i = 0, \ldots, n-1\}$ be a set of n random variables that can be ordered such that $\mathit{RV\text{-}Attributes}(y_{i-1}) \subset \mathit{RV\text{-}Attributes}(y_i)$ for $0 < i < n$. Then, the Bayesian decomposition of the conditional distribution $P(y_n \mid Z)$ is a Bayesian network (Bayesian network is defined in Section 6.4) given by:

  $$\mathit{Bayes\text{-}Decomp}(P(y_n \mid Z), \mathcal{Y}) = \{\, P(y_i \backslash\backslash y_{i-1} \mid y_{i-1}, Z) \mid 0 \le i \le n \text{with } y_{-1} = \emptyset \,\}$$

- Marginal. The marginal of $P(X \bowtie Y \mid Z)$ rooted on Y is the element of $\mathit{Bayes\text{-}Decomp}(P(X \bowtie Y \mid Z), \{Y, X \bowtie Y\}))$ with conditioned random variable Y:

  $$\mathit{Marginal}(P(X \bowtie Y \mid Z), Y) = P(Y \mid Z).$$

- Conditional. The conditional of $P(X \bowtie Y \mid Z)$ rooted on Y is the element of $\mathit{Bayes\text{-}Decomp}(P(X \bowtie Y \mid Z), \{Y, X \bowtie Y\})$ with conditioning random variable including the RV-attributes of Y:

  $$\mathit{Conditional}(P(X \bowtie Y \mid Z), Y) = P(X \backslash\backslash Y \mid Y \bowtie Z).$$

- Bayesian Compose. The Bayesian composition operation is approximately an inverse to the Bayesian decomposition. Since the argument to Bayesian compose is a Bayesian network (e.g., the output of Bayesian decompose), the operation is defined with Bayesian networks (in Section 6.4).

- Default Density. The conditional density function with respect to the Lebesgue measure of real valued domains and the counting measure for finite domains is given by $\mathit{Default\text{-}Density}(P(X \mid Y))$.

- Mean. For conditional distributions of real-valued random variables, the mean is $\mathit{Mean}(P(X \mid Y))$.

- Covariance. For conditional distributions of real-valued random variables, the covariance operator is $\mathit{Covariance}(P(X \mid Y))$.

6.3.1 Gaussian

The **Gaussian conditional distribution** for the random variable X with mean μ and covariance Σ is $P(X)$ given by the density function

$$p(X) = N(\mu, \Sigma)\colon \mathit{RV\text{-}Ran}(X) \to [0, \infty)$$

where $\mu\colon \mathit{RV\text{-}Ran}(X) \to \mathbb{R}$, $\Sigma\colon \mathit{RV\text{-}Ran}(X) \to \mathit{RV\text{-}Ran}(X)$ is a positive semidefinite operator, and

$$N(\mu, \Sigma)(x) = (2\pi|\Sigma|^2)^{-1/|X|} e^{\frac{1}{2}\langle x-\mu, \Sigma^{-1}(x-\mu)\rangle}.$$

Several standard operations for conditional distributions specialize in the case of Gaussian conditional distribution as:

- $\mathit{Conds}(P(X)) = \emptyset$
- $\mathit{Default\text{-}Density}(P(X)) = N(\mu, \Sigma)$
- $\mathit{Mean}(P(X)) = \mu$
- $\mathit{Covariance}(P(X)) = \Sigma$

Figure 6-1 and Figure 6-2 illustrate one and two dimensional Gaussian density functions. Figure 6-1 shows the general one dimensional Gaussian distribution. Figure 6-2 shows the two dimensional Gaussian distribution for random variables $\{x, y\}$ with zero mean and covariance matrix:

$$R = \begin{pmatrix} Ex^2 & Exy \\ Eyx & Ey^2 \end{pmatrix} = \begin{pmatrix} 1 & 0.9 \\ 0.9 & 1 \end{pmatrix}.$$

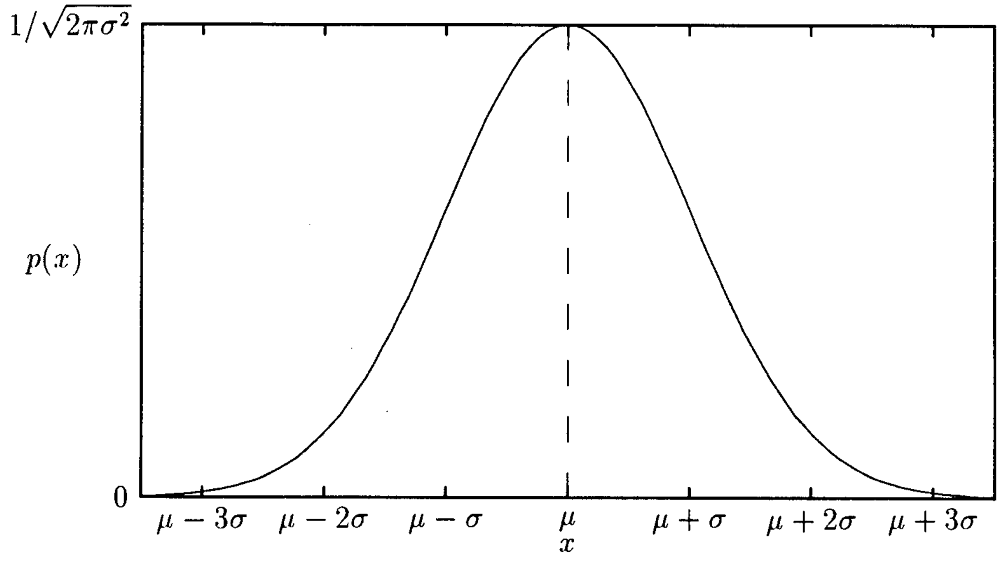


Figure 6-1: Gaussian density $p(x)$ for $x \sim N(\mu, \sigma^2)$

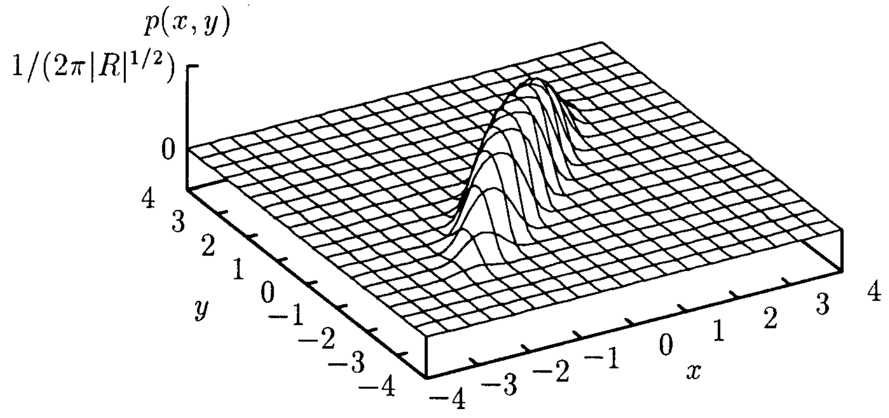


Figure 6-2: Gaussian density $p(x, y)$ for $\sigma_{xx} = 1$, $\sigma_{yy} = 1$,and $\sigma_{xy} = 0.9$

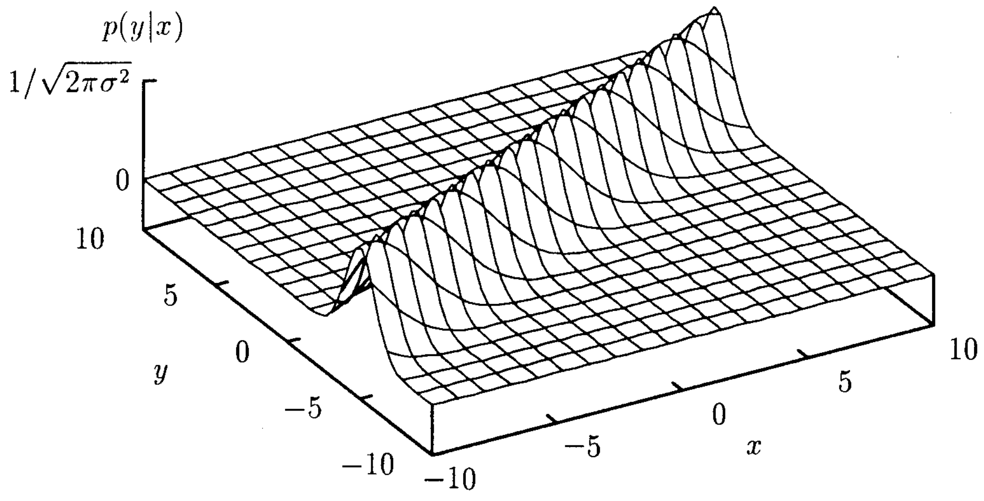


Figure 6-3: Density $p(y \mid x)$ for $y = 0.5x + u$ with $u \sim N(0, 1)$

6.3.1.1 Linear Gaussian

For real-valued Y, the **linear Gaussian conditional distribution** of X is $P(X \mid Y)$ and is a Gaussian density on X with mean as an affine function of $y \in RV\text{-}Ran(Y)$. The covariance is independent of y. A linear Gaussian conditional distribution has operations specialized as:

- $\mathit{Default\text{-}Density}(P(X \mid Y = y)) = N(A_0 + \sum_{\gamma \in Dom(y)} A(\gamma)y(\gamma), \Sigma)$
- $\mathit{Mean}(P(X \mid Y = y)) = A_0 + \sum_{\gamma \in Dom(y)} A(\gamma)y(\gamma)$
- $\mathit{Covariance}(P(X \mid Y = y)) = \Sigma$

Figure 6-3 illustrates the conditional density function associated with a linear Gaussian conditional distribution. In this case, the linear Gaussian relation is $y = 0.5x + u$ with u distributed as a zero mean unit variance Gaussian random variable. Note that the conditional density function has infinite area, clearly implying it is not the unconditional probability random variable. However, for every fixed x, the slice along y is the density for a Gaussian variable of unit variance.

6.3.2 Multi-modal Linear Gaussian

For $RV\text{-}Ran(Y) = RV\text{-}Attributes(Y) \rightarrow \mathbb{R}$ and $RV\text{-}Ran(Z)$ a discrete set of the **multi-modal linear Gaussian conditional distribution** $P(X \mid Y, Z)$ is defined such that for all $y \in RV\text{-}Ran(Y)$ and $z \in Ran(Z)$, $P(X \mid Y = y, Z = z)$ is a Gaussian density on $Ran(X)$, with mean as an affine function of y and an arbitrary function of z. The covariance depends only on z. A multi-modal linear Gaussian conditional distribution has operations specialized as:

- $Default\text{-}Density(P(X \mid Y = y, Z = z)) = N(A_{0,z} + \sum_{\gamma \in Dom(y)} A(\gamma, z) y(\gamma), \Sigma_z)$
- $Mean(P(X \mid Y = y, Z = z)) = A_{0,z} + \sum_{\gamma \in Dom(y)} A(\gamma, z) y(\gamma)$
- $Covariance(P(X \mid Y = y, Z = z)) = \Sigma_z$

6.3.3 Discrete

For the discrete conditional distribution, $P(X = x \mid Y = y)$ is a table.

6.4 Bayesian Networks

Bayesian networks organize Bayesian inference around sets of conditional distributions. This section defines a system of Bayesian network objects and operations. For a rigorous development of the general theory of conditional distribution decomposition underlying Bayesian networks, see [Loève 1978], especially pg. 26-30.

A **Bayesian network** is a constrained (and usually finite) set of conditional distributions supporting operations for computing joint distributions of all random variables. The constraint satisfied by Bayesian networks is that the conditional distributions can be ordered so that if a random attribute appears in a conditioned random variable, it does so in only one such variable, and always before appearing in any conditioning random variable. With this ordering constraint satisfied, a probabilistic chain rule can be applied to produce a unique joint distribution over all random variables.

An example of a Bayesian network is the set of conditional distributions

$$A = \{P(W, X \mid Y, Z), P(Y \mid Z), P(V \mid Y), P(Z)\}$$

where V, W, X, Y, and Z are disjoint random variables over the same probability space. Figure 6-4 shows A represented as a graph. The bipartite graph has conditional distribution nodes (triangles) connecting random variables (circles). For clarity in the

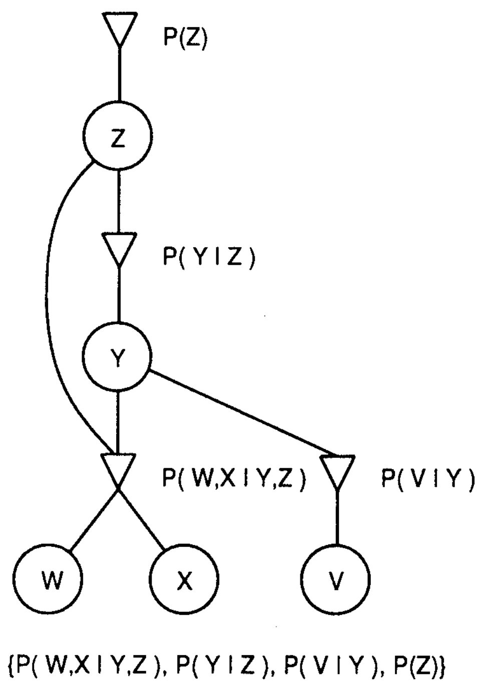


Figure 6-4: Example Bayesian network

example, each triangle node is labeled with the associated conditional distribution. This labeling is superfluous (and not used in practice) since the conditioning and conditioned random variables are the join of random variables attached to the tail and point of the triangle, respectively.

The chain rule constraint on a set of conditional distributions is equivalent to a certain relation, $\prec$, being a partial order. Here, $\prec$ is the transitive closure of the relation $<$ in conditional distributions, where $C_1 < C_2$ if some random attributes of the conditioning random variable of C_2 is an attribute of the conditioned variable of C_1. These relations lead to several operations on Bayesian networks:

- *Roots* and *Leaves*. The roots and leaves of a Bayesian network N are sets of conditional distributions given by,

  $$Roots(N) = \{\, C \mid C \in N \text{and there is no } C_1 \in N \text{ such that } C_1 < C \,\}$$

  and

  $$Leaves(N) = \{\, C \mid C \in N \text{and there is no } C_1 \in N \text{ such that } C < C_1 \,\}.$$

- *Is-Root*(N, C) and *Is-Leaf*(N, C). These test whether C is a root or leaf of N.

- *Parents*, *Ancestors*, *Children*, *Descendants*. Each conditional distribution C in a Bayesian network N has several sets of associated conditional distributions,

$$\begin{aligned} \mathit{Parents}(N,C) &= \{ C_1 \mid C_1 \in N, C_1 < C \} \\ \mathit{Ancestors}(N,C) &= \{ C_1 \mid C_1 \in N, C_1 \prec C \} \\ \mathit{Children}(N,C) &= \{ C_1 \mid C_1 \in N, C < C_1 \} \\ \mathit{Descendants}(N,C) &= \{ C_1 \mid C_1 \in N, C \prec C_1 \} \end{aligned}$$

- *Is-Parent*(N, C, C_1), *Is-Ancestor*(N, C, C_1), *Is-Child*(N, C, C_1), *Is-Descendant*(N, C, C_1). There test whether C is a parent, etc. of C_1 in N.

The operations on Bayesian networks that involve Bayesian inference are Bayesian compose, marginal Bayesian compose, substitute, observe, and posterior. These are defined by:

- *Bayesian-Compose*. Bayesian composition essentially inverts Bayesian decomposition and is defined for any Bayesian network. For the case of two conditional distributions $P(W \mid X)$ and $P(Y \mid Z)$ with Y and $W \bowtie X$ disjoint, Bayesian decomposition satisfies:

$$\begin{aligned} \mathit{Bayes\text{-}Comp}(\{P(W \mid X), P(Y \mid Z)\}) &= P(W \bowtie Y \mid (X \bowtie Z) \backslash\backslash (W \bowtie Y)) \\ &= P(W \bowtie Y \mid (X \bowtie Z) \backslash\backslash W) \end{aligned}$$

where

$$\begin{aligned} &\mathit{Bayes\text{-}Decomp}(P(W \bowtie Y \mid (X \bowtie Z) \backslash\backslash W), W) \\ &= \{P(W \mid (X \bowtie Z) \backslash\backslash W), P(Y \mid W \bowtie X \bowtie Z)\}. \end{aligned}$$

Further, the measures of $P(W \mid (X \bowtie Z) \backslash\backslash W)$ are independent of values of $Z \backslash\backslash W$ and the measures of $P(Y \mid W \bowtie X \bowtie Z)$ are independent of values of $(W \bowtie X) \backslash\backslash Z$.

For an arbitrary Bayesian network $N = \{P(X_i \mid Y_i)\}$ where X_i and $\underset{0 \leq j < i}{\bowtie} (X_j \bowtie Y_j)$ are disjoint, the Bayesian decomposition satisfies:

$$\mathit{Bayes\text{-}Comp}(N) = P(\underset{i \geq 0}{\bowtie} X_i \mid (\underset{i \geq 0}{\bowtie} Y_i) \backslash\backslash \underset{i \geq 0}{\bowtie} X_i).$$

- *Marginal-Bayes-Comp*. Marginal Bayesian composition is a convenience for organizing computations for efficiency. The operation defined in terms of other operations by:

$$\mathit{Marginal\text{-}Bayes\text{-}Comp}(N, W) = \mathit{Marginal}(\mathit{Bayes\text{-}Comp}(N), W).$$

- *Substitute*. Substitution on a Bayesian network yields another Bayesian network with one conditional distribution replaced with a given one. Usually, the original network has to go through (at least) a composition and decomposition step to derive a conditional distribution that can be simply replaced. Thus, to substitute $Q(\,X \mid Y\,)$ into a Bayesian network defining the distribution $P(X, Y, Z)$, we decompose $P(X, Y, Z)$ as

  $$P(X, Y, Z) = \mathit{Bayes\text{-}Comp}(\{P(\,Y\,), P(\,X \mid Y\,), P(\,Z \mid X, Y\,)\})$$

  and replace the $P(\,X \mid Y\,)$ with $Q(\,X \mid Y\,)$. The terms in the decomposition are:

  $$\mathit{Marginal}(P(X, Y, Z), Y)$$
  $$\mathit{Marginal}(\mathit{Conditional}(P(X, Y, Z), Y), X)$$
  $$\mathit{Conditional}(\mathit{Conditional}(P(X, Y, Z), Y), X).$$

  Thus, the defining property of substitute is:

  $$\mathit{Substitute}(P(X, Y, Z), Q(\,X \mid Y\,)) = \mathit{Bayes\text{-}Comp}(\{P(\,Y\,), Q(\,X \mid Y\,), P(\,Z \mid X, Y\,)\}).$$

  For Bayesian network computations, a Substitute is typically symbolically represented through intermediate computations until obtaining the necessary Bayesian decomposition allowing direct replacement of one conditional distribution for another.

- Observe. Observe is a special case of Substitute. It substitutes for a pure prior conditional distribution ($\mathit{Conds}() = \emptyset$) a new prior conditional distribution with probability mass 1.0 allocated to a single value of the conditioned random variable. Letting y be a value of Y and $Q_y(Y)$ be the distribution that assigns probability 1.0 to $Y = y$, then

  $$\begin{aligned} \mathit{Observe}(P(X, Y), y) &= \mathit{Substitute}(P(X,Y), Q_y(Y)) \\ &= \mathit{Bayes\text{-}Comp}(Q_y(Y), P(\,X \mid Y\,)). \end{aligned}$$

- Posterior. Bayesian network inference algorithms are designed to quickly and accurately compute posterior conditional distributions. The following computes the posterior conditional $P(\,X \mid Y, Z = z^*\,)$ from a Bayesian network N:

  $$\begin{aligned} &\mathit{Posterior}(N, X, Y, z^*) \\ &\quad = N.\mathit{Bayes\text{-}Comp}.\mathit{Observe}(z^*).\mathit{Marginal}(X \bowtie Y).\mathit{Conditional}(Y) \end{aligned}$$

  where the left associative infix notation $f(x, y) \equiv x.f(y)$ reduces nesting of parentheses.

The Symbolic Probabilistic Inference algorithm, based on set factoring (section 6.1 of [Li 1992]), efficiently processes single queries and multiple ad hoc queries. For processing an "all posterior marginals" query, a modification to emulate to operation of the Lauritzen-Spiegelhalter algorithm would further improve efficiency.

For the most general implementation of Bayesian networks, Bayesian compose would work on any combination of types of conditional distributions. Also, Bayesian decompose would be capable of producing all asked for combinations of conditional distribution types. Not providing for all combinations results in constraints on the forms of Bayesian networks allowed (such as no discrete variable following a continuous variable). If a Bayesian network contains multiple types of conditional distributions, the implementation should support Bayesian compose and decompose for those multiple types.

Algorithm efficiency partly depends on the efficiency of Boolean-like operations over random variables and conditional distributions. Such operations include: finding the join or rv-difference of random variables, and finding all conditional distributions with conditioning (or conditioned) random attributes intersecting the random attributes of other random variables. Auxiliary software structures and conventions can help streamline such operations. Potentially useful auxiliary structures and conventions include:

- Assigning unique numbers to the random attributes, random variables, and conditional distributions, and representing all sets as ordered sequences of elements.
- Storing pointers in conditional distributions software objects to predecessor and successor conditional distributions, and to conditioning and conditioned random variables.

Capabilities that may be considered later include:

- Stochastic simulation techniques
- Term computation techniques
- Network preprocessing to emulate the clique tree processing of the evidence potential algorithms.

6.5 Geometric/Observation Models

The IUE should be able to represent a scenario of multiple sensors (including cameras), multiple collections, and uncertain site models. This can require representing geometric/observation quantities with random variables and use conditional distributions to represent relations among the random variables. Differential manifolds can represent such physical quantities, and tangent approximations can represent approximate probability distributions on the physical quantities. The following subsections describe manifolds with tangent approximations, an overall modeling approach, and some specific geometric/observation models.

6.5.1 Differential Manifolds and Gaussian Tangent Approximations

IU practitioners are often directly concerned with physical objects (e.g., buildings, sensor platforms, sensors, and images), transformations between objects, and random variables of (or uncertainty in) objects and transformations. To analyze physical situations, we set up conceptual mappings from physical objects and transformations to vectors of numbers. Objects and two types of useful mappings (charts and tangent maps) from objects to vectors are illustrated in Figure 6-5. This section elaborates

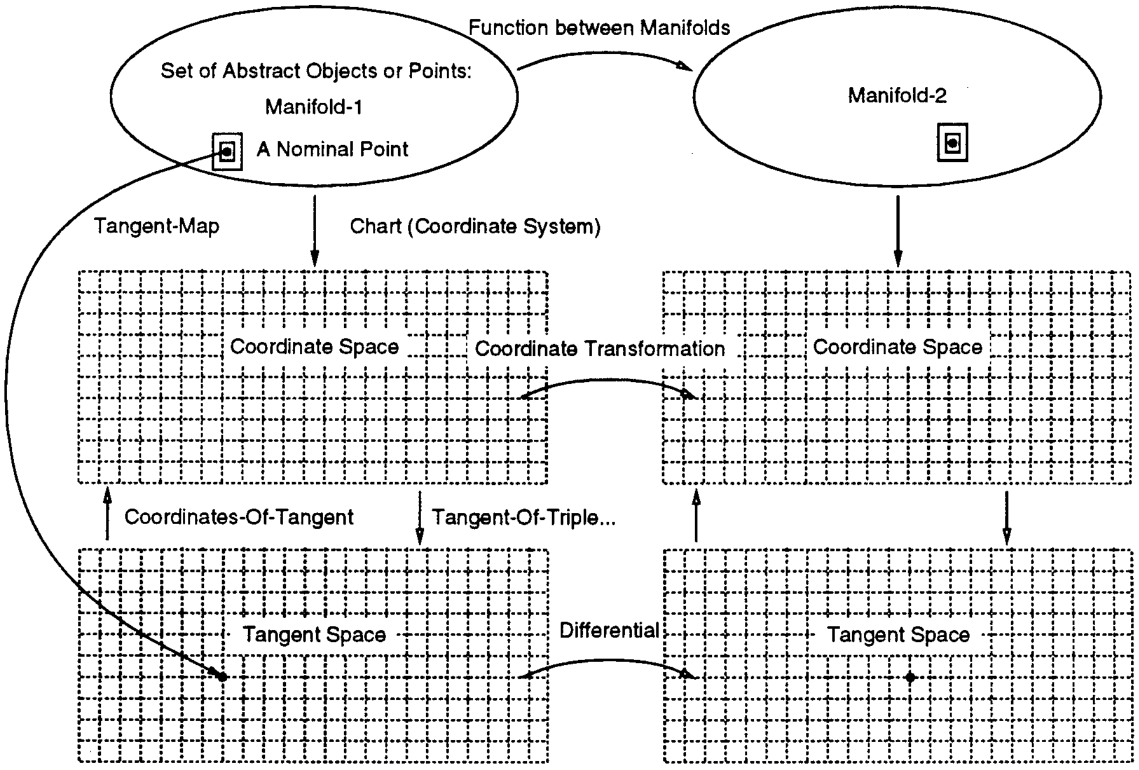


Figure 6-5: Representing abstract objects with linear spaces

on the entities depicted in the figure.

We will be describing sets of physical objects or physical situations (e.g., the set of boxes of all possible sizes, or a set of possible camera orientations) that have a topology (of open sets) and can be represented locally using vector spaces. Such a set is called a **manifold**.

One type of useful mapping of a manifold to vectors is the **chart** or **coordinate system**. Such a mapping uniquely identifies each object (or point) in an open subset of the manifold with a vector in some **Banach space** (complete normed linear space). A chart is a continuous and locally continuously invertible function from an open subset of the manifold to a Banach space (often either $\mathbb{R}^n$ or a linear space isomorphic to $\mathbb{R}^n$). Because the Banach space containing the range of a chart is often $\mathbb{R}^n$, the space is called a **coordinate space** and its vectors are called **coordinates** or **coordinate vectors**. If all coordinate spaces are isomorphic to $\mathbb{R}^n$ (i.e., the space is of finite dimensional$n$), then the manifold is said to be n-dimensional or of dimension n. For a chart φ, the exact range, $Ran(\varphi)$, is an open subset of a Banach space. In most contexts, the Banach space itself would be the value of $Range(\varphi)$.

Two charts φ_1 and φ_2 on the same manifold are compatible if $\varphi_2 \circ \varphi_1^{-1}$ is continuous and continuously invertible on $\varphi(Dom(\varphi_1) \cap Dom(\varphi_2))$. A set of pairwise compatible charts is called an **atlas**. For our applications, it is usually the case that all charts of interest for a given manifold are compatible.

As a mapping, a coordinate system (chart) represents everything we know about the "meaning" of coordinate vectors as representations for objects in a manifold. On the other hand, a coordinate space could be the range space of many charts, and is therefore without intrinsic manifold-related meaning.

A chart defines an alternate representation of points in a manifold as coordinates in a coordinate space. Similarly, a function f from a manifold with chart φ_1, to another manifold with chart φ_2, has an alternative representation as a function between the coordinate spaces of φ_1 and φ_2. This representation is called a **coordinate transformation** and is defined as $f_{\varphi_1,\varphi_2} = \varphi_2 \circ f \circ \varphi_1^{-1}$. The domain of f_{φ_1,φ_2} is $\varphi_1(Dom(\varphi_1) \cap Dom(\varphi_2))$. Also, f restricted to $Dom(\varphi_1) \cap Dom(\varphi_2)$ is $\varphi_2^{-1} \circ f_{\varphi_1,\varphi_2} \circ \varphi_1$. In general, a coordinate transformation may be nonlinear and not amenable to exact analysis.

Linearization is a powerful tool for analyzing a function between manifolds. To linearize a function between manifolds, we introduce a linear space (called a **tangent space** or **tangent vector space**), to represent an infinitesimal region of a manifold. The tangent space is independent of particular choice of chart. We also introduce a linear mapping (called a **differential**) between tangent spaces. Taken together, a differential and it's domain and range tangent spaces linearize the nonlinear situation around a specific nominal point of the original function's domain.

To complete the relation between tangent spaces and linearization, we need to introduce the operation *Derivative-Map*. This operation acts on functions mapping between Banach spaces (not between general manifolds). If f is a function from an open subset of a Banach space to another Banach space and x is an element of the domain of f, then

- *Derivative-Map*(f, x) is (if it exists) the continuous linear mapping from the Banach space containing $Dom(f)$ to the Banach space containing $Ran(f)$ such that for vectors y in the linear space containing $Dom(f)$ we have

$$\lim_{|y|\to 0} \frac{f(x+y) - (f(x) + \textit{Derivative-Map}(f,x)(y))}{|y|} = 0.$$

Each point x of a manifold has an associated tangent space, T_x. A **tangent** or **tangent vector** in T_x is modeled as an equivalence class of curves passing through x. By curve, we mean a function from an open interval in $\mathbb{R}$ containing 0 to the manifold containing x. Two curves γ_1 and γ_2 are equivalent if and only if $\gamma_1(0) = \gamma_2(0) = x$ and they pass through x with identical "direction" and "rate." That is, for some chart φ with x in its domain we have

$$\textit{Derivative-Map}(\varphi \circ \gamma_1, \gamma_1^{-1}(x)) = \textit{Derivative-Map}(\varphi \circ \gamma_2, \gamma_2^{-1}(x)).$$

If the derivative condition holds for one chart it holds for all charts.

We call the point associated with a tangent space the **nominal** point or point of linearization. Tangent spaces (and therefore tangents) are uniquely paired with the original nominal points. That is, the nominal can be uniquely recovered from a tangent space or from a tangent. It follows that tangent spaces for different nominals are always disjoint.

There is a mapping to tangents from triples where the triples have the form (x, φ, v) with x the nominal point in the manifold, φ a chart with x in its domain, and v a point in the coordinate space of φ. Two triples (x_1, φ_1, v_1) and (x_2, φ_2, v_2) map to the same tangent if and only if $x_1 = x_2$ and $\textit{Derivative-Map}(\varphi_2 \circ \varphi_1^{-1}, \varphi_1(x))(v_1) = v_2$. If φ is a chart with x in its domain, the tangent space at x is isomorphic to the coordinate space containing $Ran(\varphi(x))$. A triple (x, φ, v) can be thought of as representing a "φ-linear" curve, γ, satisfying $\gamma(h) = \varphi^{-1}(\varphi(x) + hv)$ for small $h \in \mathbb{R}$. The tangent vector of the triple is the equivalence class of γ.

Operations relating combinations of manifolds, charts, coordinates, tangents, and tangent spaces include:

- *Nominal-Of-Tangent-Space*$(T) = x$, the nominal point for tangent space T_x.
- *Nominal-Of-Tangent*$(t) = x$, the nominal point for the tangent t. (In the theory of differential manifolds, this operation is the projection onto the base of a tangent bundle.)
- *Nominal-Coordinates-Of-Tangent-Space*$(\varphi, T_x) = \varphi(x)$, where $x =$ *Nominal-Of-Tangent-Space*(T_x).

- *Nominal-Coordinates-Of-Tangent*$(\varphi, t) = \varphi(x)$, where $x =$ *Nominal-Of-Tangent*(t).
- *Coordinates-Of-Tangent*$(\varphi, t) = v$ where (x, φ, v) represents tangent t.
- *Offset-Point-Of-Tangent*$(\varphi, t) = \varphi^{-1}(\varphi(x)+v)$ where $x =$ *Nominal-Of-Tangent*(t) and $v =$ *Coordinates-Of-Tangent*(φ, t). (Note that the indicated φ^{-1} may not exist for all combinations of φ and t.) For "small" tangents, this operation becomes independent of choice of chart. The operation represents the linearization approximation in an infinitesimal region of the nominal.
- *Offset-Coordinates-Of-Tangent*$(\varphi, t) = \varphi(x)$ where $x =$ *Offset-Point-Of-Tangent*(φ, t).
- *Tangent-Space-Of-Tangent*$(t) = T_x$ where T_x is the tangent space containing tangent t.
- *Tangent-Space-Of-Nominal*$(X, x) = T_x$ the tangent space for manifold X at point $x \in X$.
- *Tangent-Of-Triple-With-Nominal*$(x, \varphi, v) = t$, the tangent represented by (x, φ, v) as described above.
- *Tangent-Of-Triple-With-Space*$(T, \varphi, v) = t$, the tangent represented by (x, φ, v) where $x =$ *Nominal-Of-Tangent-Space*(T).
- *Default-Chart-Of-Tangent-Space*$(T) =$ some chart φ such that for any tangent t in tangent space T_x,
  (*Point-Of-Tangent*$(t), \varphi,$ *Default-Point*(t)) represents t.
- *Default-Chart-Of-Tangent*$(t) =$ some chart φ such that
  (*Point-Of-Tangent*$(t), \varphi,$ *Default-Point*(t)) represents the tangent t.
- *Default-Coordinates-Of-Tangent*$(t) = v$, where v is the coordinate vector in the coordinate space of $\varphi =$ *Default-Chart-Of-Tangent*(t) such that (x, φ, v) represents t.
- *Default-Offset-Coordinates-Of-Tangent*$(t) = \varphi(x)$, where $\varphi =$ *Default-Chart-Of-Tangent*(t) and $x =$ *Offset-Point-Of-Tangent*(φ, t).
- *Default-Manifold-Of-Point*$(x) = X$, where X is some manifold containing x.
- *Default-Nominal-Coordinates-Of-Tangent*$(t) = \varphi(x)$ where $x =$ *Nominal-Of-Tangent*(t) and $\varphi =$ *Default-Chart-Of-Tangent*(t).
- *Default-Nominal-Coordinates-Of-Tangent-Space*$(T) = \varphi(x)$ where $x =$ *Nominal-Of-Tangent-Space*(T) and $\varphi =$ *Default-Chart-Of-Tangent-Space*(T).

- *Differential-Map*(f,x) = the differential at point x of the function $f\colon X \to Y$ between manifolds X and Y. The differential (if it exists) is a continuous linear function from T_x = *Tangent-Space-Of-Nominal*$(Dom(f),x)$ to $T_{f(x)}$ = *Tangent-Space-Of-Nominal*$(Ran(f), f(x))$ such that if γ is a curve in the equivalence class of a tangent $t \in T_x$, then $f \circ \gamma$ is a curve in the tangent vector given by *Differential-Map*$(f,x)t$. Equivalently, for all tangent vectors $y \in T_{f(x)}$ and charts φ_1 on x and φ_2 on $f(x)$ we have

$$\begin{aligned}&\textit{Differential-Map}(f,x)(y)\\&\quad = \textit{Tangent-Of-Triple-With-Point}(f(x), \varphi_2, \textit{Derivative-Map}(\varphi_2 \circ f \circ \varphi_1^{-1}, \varphi_1(x))\\&\qquad(\textit{Coordinates-Of-Tangent}(\varphi_1, y)))\end{aligned}$$

The operations (excluding derivatives and differentials) are summarized in Table 6-1. In the table, entries with "i" or "o" indicate types of inputs or outputs, respectively, to a given operation.

Table 6-1: Tangent space related operations

Operation	X	φ	x	v	T	t
Nominal-Of-Tangent-Space			o		i	
Nominal-Of-Tangent			o			i
Nominal-Coordinates-Of-Tangent-Space			o		i	
Nominal-Coordinates-Of-Tangent			o			i
Coordinates-Of-Tangent		i		o		i
Offset-Point-Of-Tangent		i	o			i
Offset-Coordinates-Of-Tangent		i		o		i
Tangent-Space-Of-Tangent					o	i
Tangent-Space-Of-Nominal	i		i		o	
Tangent-Of-Triple-with-Nominal		i	i	i		o
Tangent-Of-Triple-with-Space		i		i	i	o
Default-Chart-Of-Tangent-Space		o			i	
Default-Chart-Of-Tangent		o				i
Default-Coordinates-Of-Tangent				o		i
Default-Offset-Coordinates-Of-Tangent				o		i
Default-Manifold-Of-Point	o		i			
Default-Nominal-Coordinates-Of-Tangent				o		i
Default-Nominal-Coordinates-Of-Tangent-Space				o	i	

A random variable may have its range in a manifold and have approximating random variables on the manifold's tangent spaces. Suppose $X\colon \Omega \to M$ is a random variable taking on values in a manifold M (assuming open sets in M are measurable) and that a random variable X_m exists in the tangent space of M at $m \in M$ such that for any bounded set S in the coordinate space the distribution of X_m satisfies

$$\lim_{\delta \to 0} \frac{P(\{\varphi^{-1}(\varphi(m)+y) \mid y \in \delta S\}) - \int_{y \in \delta S} dy N(0,\Sigma)(y)}{\delta^2} = 0.$$

We call any random variable such as X_m a **Gaussian tangent approximation** of X at m.

If $f \colon M \to N$ is a differentiable function between manifolds, X is a random variable on M, X_m is a Gaussian tangent approximation of X at m, then a Gaussian tangent approximation of $f \circ X$ at $f(m) \in N$ is $Y_{f(m)}$ defined by

$$Y_{f(m)} = \textit{Differential-Map}(f,m) X_m.$$

6.5.2 Modeling Approach

A modeling approach based on the above concepts is to:

- Identify physical and abstract objects (or points) of interest. Points include physical structures (boxes, polygonal shapes, ellipses, etc.), frames, exterior or interior orientations of a camera, images, and image features. More abstract objects include point configurations, transformations between frames, and sensor projections. Normally each class of interesting points forms a manifold (with at least one chart described in terms of physical items).

- Identify random variables taking on points as values. There may be uncertainty concerning the size or orientation of a box, the external orientation or internal orientation of a camera, or the location of an extracted image feature. Such uncertainties are represented with random variables. Instead of directly analyzing these variables and their distributions, we nearly always form and analyze Gaussian tangent approximations.

- Identify the functions (and corresponding conditional distributions) relating points and random variables. And collect them into a scene (set of such functions). The functions may be that a box face is parallel in the plane of the ground, the box is 20 to 30 meters long, the projection of 3-space location depends on camera orientation, etc. Transformations between physical and or image spaces include: rigid motion, translation, rotation, scaling, general linear transformation, general affine transformation, orthogonal projection, perspective projection, etc.

- Identify the charts (or coordinate systems) that map points into $\mathbf{R}^n$ or other Banach spaces. With charts, each set of points becomes a manifold. Charts may represent such notions as a box's shape as described by the triple of length, width, and height in meters, shape described by diagonal distances in inches, or a camera's external orientation being described by three rotation angles and three offsets $[\omega\, \phi\, \kappa\, X\, Y\, Z]$ with rotations and translation performed in a particular order and with respect to a particular frame of reference.

- Recognize that to each point in a manifold there corresponds a tangent space. Given a point and a chart, the Banach space containing the range of the chart is isomorphic to the tangent space.

- Identify Gaussian tangent approximation random variables in the tangent spaces of manifolds. These approximate the random variables in the original spaces. Wherever original random variables on manifolds are functionally related to another random variable, relate the corresponding Gaussian tangent approximations by the differential of the function between manifolds.

- Collect the conditional distributions of the jointly Gaussian tangent approximations of the functions in the scene into a Bayesian network, attach available evidence (using *Observe*), and solve queries using a Bayesian network inference algorithm. Use the posterior mean values to select new nominals for relinearization. Repeatedly relinearize and resolve until solutions converge with respect to posterior covariances.

- As necessary, use the operators of Table 6-1 to map between original manifolds, coordinate spaces, tangent approximations, and other entities discussed in Section 6.5.1.

A **scene** represents a scenario of image collections. It denotes the physical things of interest and their causal relations (both deterministic and stochastic). Formally, a scene is a Bayesian network. The random attributes denote various degrees of freedom that span the variations found in different realizations of the collection scenarios. The conditional distributions represent causal relations between the physical things denoted by the random attributes.

RV-Ran's of random variables of scenes may contain many types of objects, including: physical objects, coordinate spaces, tangent spaces, transformations, sensors, collection events, images, and extracted features.

A scene supports operations beyond those for ordinary Bayesian networks. Additional operations include those for restricting attention to differentials between tangent spaces and for extracting and manipulating Bayesian networks on tangent spaces.

6.5.3 Manifolds of Interesting Objects and Transformations

This section describes the several manifolds of physical objects and transformations of general interest for IU. Not all manifolds require an explicit software class to be represented. For instance, a manifold of triangles can be described within the manifold of polygons without creating a new class. However, it may be possible to more efficiently implement triangles as a separate class. It is also possible to arrive at many particular spatial and image transformations from a general affine transformation.

The following paragraphs list the manifolds in rough order of: objects free to move in space, objects constrained to move with the earth, transformations between objects, sensor projections, image objects, and transformations between image objects.

Spatial Point A **spatial point** is smoothly varying space-time trajectory. Although not necessary, here and throughout we assume a Euclidean space-time (i.e., we assume all relative velocities are small and gravity is weak). The set of spatial points is an infinite dimensional manifold with coordinate spaces isomorphic to the space of smooth functions from $(0,1)$ to $\mathbb{R}^3$. An interesting special case of spatial points might have them constrained to move rigidly with some given object such as the earth or sensor.

Spatial Point n-Set A **spatial point n-set** is a set of n spatial points. For fixed n, the set of spatial point n-sets forms an infinite dimensional manifold with coordinate spaces isomorphic to the space of smooth functions from $(0,1)$ to $\mathbb{R}^{3n}$. An interesting special case of Spatial Point n-Set might have the constituent spatial points constrained to move rigidly with some given object such as the earth or a sensor.

Spatial Point n-Tuple A **spatial point n-tuple** is a tuple mapping n attributes into spatial points (not necessarily distinct). The set of attributes is usually $\{0, 1, \ldots, n-1\}$, but arbitrary labelings of points are allowed. For fixed n, the set of spatial point n-tuples forms a $3n$-dimensional manifold. Any transformation that acts on spatial points has an induced version (page 25) that acts on spatial point n-tuples. An interesting special case of Spatial Point n-Tuple might have the constituent spatial points constrained to move rigidly with some given object such as the earth or a sensor.

Rigid Spatial Point n-Set A **rigid spatial point n-set** is a spatial point n-set with elements that could be affixed (with no sliding) to some rigid object undergoing some smooth rigid motion. For fixed n, the set of rigid n-sets forms an infinite dimensional manifold with coordinate spaces isomorphic to the product space $\mathbb{R}^{k_n} \times ((0,1) \xrightarrow{\text{smooth}} \mathbb{R}^{3n-k_n}$ where k_n is given by:

n (Points)	k_n (Dimension)
1	0
2	1
$n \geq 3$	$k_n = 3n - 6$

An interesting special case only deals with rigid spatial point n-sets constrained to move rigidly with some given object such as the earth or a sensor.

Rigid Spatial Point n-Tuple A **rigid spatial point n-tuple** is a spatial point n-tuple with domain elements that could be affixed (with no sliding) to some rigid object undergoing some smooth rigid motion. For fixed n, the set of rigid n-tuples forms an infinite dimensional manifold with coordinate spaces isomorphic to the product space $\mathbb{R}^{k_n} \times ((0,1) \xrightarrow{\text{smooth}} \mathbb{R}^{3n-k_n}$ where k_n is as given above. An interesting special case only deals with rigid spatial point n-tuple constrained to move rigidly with some given object such as the earth or a sensor.

n-Configuration An **n-configuration** is an equivalence class of rigid spatial point n-sets. Two rigid spatial point n-sets are equivalent if there is some moving rigid body that could "transfer" the points of one set at one time to coincide with the points of the other set at some time. To avoid dealing with manifolds with boundary, we further restrict n-configurations with $n \geq 3$ to have no symmetries under rotation and translation of the rigid body. The set of n-configurations forms a k_n-dimensional manifold with k_n as given above for rigid spatial point n-sets.

Earth-Fixed Point An **earth-fixed point** is a spatial point that, over time, maintains a constant offset from every spatial point affixed to solid earth. The constant offset is one linear combination of vectors connecting, over time, some distinguished locations on (or in) solid earth. The set of earth-fixed points forms a 3-dimensional manifold. Typically, earth-fixed points will be "produced" by chains of transformations applied to earth-fixed frames.

Given a reference spheroid, a **geodetic coordinate system** is a chart that maps earth-fixed points to height above the spheroid, latitude, and longitude.

Earth-Fixed n-Set An **earth-fixed n-set** is a set of n earth-fixed points. For fixed n, the set of earth-fixed n-sets forms a $3n$-dimensional manifold.

Earth-Fixed n-Tuple An **earth-fixed n-tuple** is a rigid spatial point n-tuple where the range elements are earth-fixed points. For fixed n, the set of earth-fixed n-tuples forms a $3n$-dimensional manifold.

Earth-Fixed Frame An **earth-fixed frame,** τ, is an earth-fixed 4-tuple having attributes $\{0, 1, 2, 3\}$ such that the vectors $\tau_1 - \tau_0$, $\tau_2 - \tau_0$, and $\tau_3 - \tau_0$ form a right-handed orthonormal sequence. The earth-fixed point τ_0 is called the origin. The vector $\tau_i - \tau_0$ is called the x-, y-, and z-axis for $i = 1$, 2, and 3, respectively. The set of earth-fixed frames forms a 6-dimensional manifold.

Geocentric Frame A **geocentric frame** is an earth-fixed frame with origin at the center of a reference spheroid, z-axis directed toward the north pole, x-axis such that the x-z plane contains the Greenwich meridian. Given a reference spheroid, there is just one geocentric frame (i.e., the set consisting of that frame forms a 0-dimensional manifold).

Local Vertical Frame A **local vertical frame,** is an earth-fixed frame with z-axis (local up) pointing away from the center of a reference spheroid and x-axis (local east) pointing along the cross product the astronomical north vector with the z-axis. The y-axis then points to local north. Given a reference spheroid, the set of local vertical frames forms a 3-dimensional manifold.

Spatial Line A **spatial line** is a doubly infinite set of points as would be obtained by extending the x-axis of a spatial frame. The set of all spatial lines is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \xrightarrow{\text{smooth}} \mathbb{R}^4$.

Spatial Ray A **spatial ray** is a singly infinite set of points as would be obtained by extending the positive x-axis of a spatial frame. The set of all spatial rays is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \xrightarrow{\text{smooth}} \mathbb{R}^5$.

Spatial Line Segment A **spatial line segment** is a bounded set of points as would be obtained by choosing an interval along x-axis of a spatial frame. The set of all spatial line segments is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \xrightarrow{\text{smooth}} \mathbb{R}^6$.

Spatial Plane A **spatial plane** is the set of points as would be obtained by taking all affine combinations of points on x- and y-axes of a spatial frame. The set of all spatial planes is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \xrightarrow{\text{smooth}} \mathbb{R}^3$.

Spatial Planar n-Polygon A **spatial planar polygon** is a planar rigid spatial point set that consists of a n vertices and connecting line segments. Segments are disjoint except that every vertex has exactly two abutting line segments. The set of all spatial planar polygons with angles other than 0° or 180° at each vertex is an infinite dimensional manifold with coordinate spaces isomorphic to $\mathbb{R}^{2n-3} \times (0,1) \overset{\text{smooth}}{\longrightarrow} \mathbb{R}^6$.

Spatial Triangle A **spatial triangle** is a spatial planar 3-polygon.

Spatial Rectangle A **spatial rectangle** is a spatial planar 4-polygon with all right angles.

Spatial Square A **spatial square** is a spatial rectangle with equal length sides.

Spatial Vector A **spatial vector** is the equivalence class of differences between points that could be connected by parallel transport across a rigid object. The set of all spatial vectors is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \overset{\text{smooth}}{\longrightarrow} \mathbb{R}^3$.

Spatial Direction A **spatial direction** is a spatial vector of unit length. The set of all spatial directions is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \overset{\text{smooth}}{\longrightarrow} \mathbb{R}^2$.

Spatial Box A **spatial box** is a rigid rectangular box (6-faced polyhedron with all right angles) with a smooth trajectory in space-time. A spatial box has a tuple of four distinguished vertices, the first vertex adjacent to the remaining three. If τ is the tuple of vertices, then τ_0 is called the origin and τ_i is called the x-, y-, and z-vertex for $i = 1$, 2, and 3, respectively. Also, $\tau_i - \tau_0$ is called the x-, y-, and z-edge vector and the ray in that direction is called the x-, y-, and z-axis. Also, the tuple of axes is assumed to form a right-handed system. The set of spatial boxes forms an infinite dimensional manifold with coordinate spaces isomorphic to $\mathbb{R}^3 \times ((0,1) \overset{\text{smooth}}{\longrightarrow} \mathbb{R}^6$.

Half-Infinite Spatial Box A **half-infinite spatial box** is a rigid rectangular box, one finite rectangular face, and four half-infinite rectangular faces, leaving the final finite face to exist at infinity. The box is assumed to have a smooth trajectory in space-time. A half-infinite spatial box has a tuple of three distinguished vertices, the first vertex adjacent to the remaining two. If τ is the tuple of vertices, then τ_0 is called the origin and τ_i is called the x- and y-vertex for $i = 1, 2$ respectively. The rays from the origin to the α-vertex is called the α-axis, for $\alpha =$ "x and "y". The ray from the origin along the infinite length of the box is called the z-axis. Finally, the tuple of axes is assumed to form a right-handed system. The set of half-infinite spatial boxes forms an infinite dimensional manifold of coordinate spaces isomorphic to $\mathbb{R}^2 \times ((0,1) \xrightarrow{\text{smooth}} \mathbb{R}^6$.

Frame Coordinate System A **frame coordinate system** is a chart on the space of points spanned by the frame vertices. The chart maps a point to its Cartesian (x, y, z) coordinates with respect to the frame. Given a frame, there is one frame coordinate system (i.e., the set consisting of that chart forms a 0-dimensional manifold).

Frame-to-Point Transformation A **frame-to-point transformation** maps a spatial frame to the spatial point at a given offset from the frame's origin. The offset is a fixed linear combination of the frame's axes. Given an offset, the set of all frame-to-point transformations is 0-dimensional (a single element).

Spatial Translation Given a spatial frame, a **spatial translation** is a translation of all spatial points with respect to the frame. For a given spatial frame, the set of spatial translations forms a 3-dimensional manifold.

Spatial Rotation Given a spatial frame, a **spatial rotation** is a rotation of all spatial points with respect to the origin of the frame. For a given spatial frame, the set of spatial rotations forms a 3-dimensional manifold.

Spatial Rigid Motion Given a spatial frame, a **spatial rigid motion** is a rigid motion of all spatial points with respect to the frame. For a given spatial frame, the set of spatial rigid motions forms a 6-dimensional manifold.

Spatial Directional Scaling Given a spatial frame, a **spatial directional scaling** is a scaling in one direction of all spatial points with the expansion centered at the origin of the frame. For a given spatial frame, the set of spatial directional scalings forms a 3-dimensional manifold.

Spatial Linear Transformation A **spatial linear transformation** is a composition of a spatial rotation and spatial directional scalings, all defined with respect to a common origin. For an origin, the set of spatial linear transformations forms a 9-dimensional manifold.

Spatial Affine Transformation A **spatial affine transformation** is a composition of a spatial linear transformation and spatial translation. The set of spatial affine transformations forms a 12-dimensional manifold.

Frame-to-Box Transformation A **frame-to-box transformation** maps a spatial frame into a spatial box with corresponding frame and box axes parallel and with frame origin at a fixed point with respect to the box. The fixed point is located as an offset from the box origin given by a linear combination of edges vectors (unnormalized). The coefficients of the offset are assumed to be given parameters of any manifold of frame-to-box transforms. Given the offset coordinates of frame origin, the set of frame-to-box transformations is a 3-dimensional manifold. The default coordinate system has coordinates corresponding to box length, width, and height.

Box-to-Frame Transformation A **box-to-frame transformation** maps a spatial box into a spatial frame with corresponding box and frame axes parallel, with frame origin at a fixed point with respect to the box. The fixed point is located as an offset from the box origin given by a linear combination of edges vectors (unnormalized). The coefficients of the offset are assumed to be given parameters of any manifold of box-to-frame transforms. Given the offset coordinates of frame origin, the set of box-to-frame transformations is a 0-dimensional manifold (a completely specified, projection-like mapping that projects off the shape of the box leaving only a frame).

Spatial Ellipsoid A **spatial ellipsoid** is a rigid ellipsoid with a smooth trajectory in space-time. An ellipsoid has a tuple of four distinguished spatial points located at the centroid and each intersection of principal axis with the surface of the ellipsoid. If τ is the tuple of vertices, then τ_0 is called the origin and τ_i is called the x-, y-, and z-surface point for $i = 1$, 2, and 3, respectively. Also, $\tau_i - \tau_0$ is called the x-, y-, and z-vector (principal axis vectors) and the ray in that direction is called the x-, y-, and z-axis. Also, the tuple of axes is assumed to form a right-handed system. The set of spatial ellipsoids forms an infinite dimensional manifold with coordinate spaces isomorphic to $\mathbb{R}^3 \times ((0,1) \xrightarrow{\text{smooth}} \mathbb{R}^6$.

Frame-to-Ellipsoid Transformation A **frame-to-ellipsoid transformation** maps a spatial frame into a spatial ellipsoid with corresponding frame and ellipsoid axes parallel and with frame origin at a fixed point with respect to the ellipsoid. The fixed point is located as an offset from the ellipsoid origin given by a linear combination of principal axis vectors (unnormalized). The coefficients of the offset are assumed to be given parameters of any manifold of frame-to-ellipsoid transforms. Given the offset coordinates of frame origin, the set of frame-to-ellipsoid transformations is a 3-

dimensional manifold. The default coordinate system has coordinates corresponding to the first, second, and third principal axes of the ellipsoid.

Ellipsoid-to-Frame Transformation An **ellipsoid-to-frame transformation** maps a spatial ellipsoid into a spatial frame such that corresponding ellipsoid and frame axes are parallel and the frame origin is at a given offset from the centroid of the ellipsoid. The offset from the centroid is given by a linear combination of principal axis vectors (unnormalized). The coefficients of the offset are assumed to be given parameters of any manifold of ellipsoid-to-frame transforms. Given the offset coordinates of frame origin, the set of ellipsoid-to-frame transformations is a 0-dimensional manifold (a completely specified, projection-like mapping that projects off the shape of the ellipsoid leaving only a frame).

Image Point An **image point** is a position in a sensor's image. The set of all image points is typically a 2-dimensional manifold.

Image Point n-Set An **image point n-set** is a set of n image points. The set of all image point n-sets is typically a $2n$-dimensional manifold.

Image Point n-Tuple An **image point n-set** is a tuple with n attributes mapping to image points. Usually, the attribute set will be $\{0, 1, \ldots, n-1\}$, but arbitrary n-sets are allowed. The set of all image point n-tuples is typically a $2n$-dimensional manifold.

Image Line An **image line** is a straight line in a sensor's image. The set of all image lines is typically a 2-dimensional manifold.

Image Ray An **image ray** is a half line in a sensor's image. The set of all image rays is typically a 3-dimensional manifold.

Image Line Segment An **image ray** is a bounded line segment in a sensor's image. The set of all image rays is typically a 4-dimensional manifold.

Image Polygon An **image polygon** is a polygon in a sensor's image. The set of all image polygons with n vertices is typically a $2n$-dimensional manifold.

Image Triangle An **image triangle** is an image polygon with three sides. The set of all image triangles is typically a 6-dimensional manifold.

Image Quadrilateral An **image quadrilateral** is an image polygon with three sides. The set of all image quadrilaterals is typically an 8-dimensional manifold.

Image Ellipse An **image ellipse** is an ellipse in a sensor's image. The set of all image ellipses is typically a 5-dimensional manifold.

Sensor A **sensor** as a function from the product of time, sensor exterior orientation, sensor interior orientation, and scene to an image or a set of features extracted from an image. A sensor transform is the Curry of a sensor with time, and exterior/interior orientations.

Sensor Exterior Orientation For a given collection, the **sensor exterior orientation** is the orientation of a physical sensor at midpoint of the sensing interval. The set of sensor exterior orientations is a 6-dimensional manifold.

Sensor Exterior Orientation Trajectory For a given collection, the **sensor exterior orientation trajectory** is the orientation of a physical sensor throughout the sensing interval. The set of sensor exterior orientations is an infinite dimensional manifold with coordinate spaces isomorphic to $(0,1) \xrightarrow{\text{smooth}} \mathbb{R}^6$. By assuming a parameterized functional form for the trajectory, the dimensionality of the set of exterior orientations can be made finite.

Sensor Interior Orientation A **sensor interior orientation** is the projection of spatial location relative to the sensor exterior orientation into image location at the midpoint of the sensing interval. In camera photography, it is sometimes assumed that sensor interior orientation is fixed for a particular camera and the set of sensor orientations is a 4-dimensional manifold (focal length, rotation, and two translations). For non-camera sensors, the interior orientation may vary between collections and depend on additional parameters.

Sensor Interior Orientation Trajectory A **sensor interior orientation** is the projection of spatial location relative to the sensor exterior orientation into image location throughout the sensing interval. The dimensionality of the set of all sensor interior orientation trajectories can be finite or infinite, depending on the assumed parameterization.

Orthographic Projection An **orthographic projection** is a particular sensor interior orientation. Given a sensor exterior orientation, the set of all orthographic projections is a manifold of dimensional 4 or less. The 4 degrees of freedom (some of which may be considered fixed) involve two translations and two scalings. An orthographic camera model might reduce degrees of freedom to three by assuming the two scalings are equal. For a given camera, it may be useful to assume that scalings are known exactly.

Perspective Projection A **perspective projection** is a particular sensor interior orientation. Given a sensor exterior orientation, the set of all perspective projections is a manifold of dimensional 4 or less. The 4 degrees of freedom (some of which may be considered fixed) involve two translations, two scalings (or a directional scaling and a focal length). A pinhole camera model might select out three degrees of freedom to model (two translations and focal length). For a given camera, it may be useful to assume that focal length and scaling are known exactly.

Orthographic SAR Projection An **orthographic SAR projection** is a particular sensor interior orientation. Given a sensor exterior orientation trajectory (including velocity vector), the set of all orthographic SAR projections is a manifold of dimensional 4 or less. The 4 degrees of freedom (some of which may be considered fixed) involve two translations and two scalings. For SAR, the scalings may often be assumed to be known exactly.

Cylindrical SAR Projection A **cylindrical SAR projection** is a particular sensor interior orientation. Given a sensor exterior orientation trajectory (including velocity vector), the set of all cylindrical SAR projections is a manifold of dimensional 4 or less. The 4 degrees of freedom (some of which may be considered fixed) involve two translations and two scalings. For SAR, the scalings may often be assumed to be known exactly.

Swept Perspective Projection A **Swept Perspective Projection** is a particular sensor interior orientation. Given a sensor exterior orientation trajectory, the set swept perspective projections is a manifold of dimension 3 or less. The 3 degrees of freedom involve two translations and one scaling (or focal length). For a given sensor, it may be useful to assume that focal length (or equivalent scaling) is known exactly.

Ellipsoid to Extreme Value Transformation The **ellipsoid to extreme value transformation** maps an ellipsoid and projection to extreme points on the projected ellipse. There is only one such mapping.

Image Translation Given an image, an **image translation** is a translation of all image points with respect to the image axes. For a given image, the set of image translations forms a 2-dimensional manifold.

Image Rotation Given an image and center of rotation, an **image rotation** is a rotation of all image points about the center. For a given image and center, the set of image rotations forms a 1-dimensional manifold.

Image Rigid Motion An **image rigid motion** is a composition of an image translation and image rotation. The set of image rigid motions forms a 3-dimensional manifold.

Image Directional Scaling An **image directional scaling** scaling in one direction of all spatial points with the expansion centered at the origin of the image. For a given image, the set of image directional scalings forms a 2-dimensional manifold.

Image Linear Transformation An **image linear transformation** is a composition of an image rotation and directional scalings, all defined with respect to a common origin. For a given origin (in a given image), the set of image linear transformations forms a 4-dimensional manifold.

Image Affine Transformation An **image affine transformation** is a composition of an image linear transformation and image translation. For a given image, the set of image affine transformations forms a 6-dimensional manifold.

Sun Centroid The **sun centroid** is the spatial point (0-dimensional) of the centroid of the sun.

Earth Centroid The **earth centroid** is the spatial point (0-dimensional) of the centroid of the sun.

Earth Orbit An **earth orbit** is the trajectory of a free falling object. The set of all earth orbits forms a 6-dimensional manifold.

Image Polynomial Warping An **image polynomial warping** is a polynomial image-to-image transformation. The dimension of the manifold of warpings is the terms in the polynomial representation of the warps.

References

[Bullock 1991] Bullock, M., Miltonberger, T., Reinholdtsen, P., Wilson, K., "SAR and IR Data Fusion Using the Sensor Algorithm Research Expert System (SARES)," *Proceedings of the SPIE Conference on Object Recognition.* Orlando, Florida. 1991.

[Chang 1991] Chang, K., Fung, R., "Symbolic Probabilistic Inference with Continuous Variables." *Uncertainty in Artificial Intelligence, Proceedings of the Seventh Conference.* Morgan Kaufman. 1991.

[D'Ambrosio] D'Ambrosio, B. Incremental Probabilistic Inference. Department of Computer Science, Oregon State University, Corvallis, Oregon.

[Ettinger 1991] Ettinger, G., Morgan, D., Reinholdtsen, P., "Bayesian Inference for Model-Based Vision'." Advanced Decision Systems. 1991.

[Fleming 1977] Fleming, W. "Functions of Several Variables." Springer-Varlag. 1977.

[Jensen 1990] Jensen, F. V., "Calculation in HUGIN of Probabilities for Specific Configuration - a Trick with Many Application". Institute for Electronic Systems, Department of Mathematics and Computer Science, Aalborg University, Aalborg, Denmark. Feb 1990.

[Lange 1985] Lange, S. "Differential Manifolds." Springer-Verlag. 1985

[Lauritzen 1988] Lauritzen, S. L., and Wermuth, N., Graphical Models for Associations Between Variables, some of which are Qualitative and some Quantitative. Aalborg University and University of Mainz, *The Annals of Statistics*, Vol. 17, No. 1, 31-57. Jun 1988.

[Lauritzen 1990] Lauritzen, S. L., Propagation of Probabilities, Means and Variances in Mixed Graphical Association Models, Aalborg University, Aalborg, Denmark. Apr 1990.

[Li 1992] Li, Z., D'Ambrosio, B., "Efficient Inference in Belief Networks as a Combinatorial Optimization Problem," Department of Computer Science Oregon State University, Oregon, Nov 1992.

[Loève 1977] Loève, M. "Probability Therory I." Springer-Verlag. 1977.

[Loève 1977] Loève, M. "Probability Therory II." Springer-Verlag. 1978.

[Olesen 1991] Olesen, K. G., Casual Probabilistic Networks with Both Discrete and Continuous Variables. ISSN 0106-0791. Aug 1991.

[Meyer 1990] Meyer, B., "Object-oriented Software Construction." Prentice Hall. 1990.

[Morgan 1989] Morgan, D., Miltonberger, T., Orr, G., "A Sensor Algorithm Expert System," *SPIE/SPSE Symposium on Electronic Imaging; Advanced Devices and Systems.* Los Angeles, California. 1989.

[Neapolitan 1990] Neapolitan, R. "Probabilistic Reasoning in Expert Systems, Theory and Algorithms." John Wiley & Sons. 1990.

[Pearl 1988] Pearl, J. "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference." Morgan Kaufman. 1988.

[Ritter 1990] Ritter, G. "Image Algebra: An Overview," *Computer Vision, Graphics, and Image Processing* **49**, 297-331, 1990.

[Shachter 1989] Shachter, R. D., and Peot, M. A., Simulation Approaches to General Probabilistic Inference on Belief Networks. Stanford University, Stanford, California. 1989.

[Shachter 1990] Shachter, R. D., Del Favero, B. A., and D'Ambrosio, B. Symbolic Probabilistic Inference: A Probabilistic Perspective. Engineering-Economic Systems, Stanford University, Stanford, California and Dept. of Computer Science, Oregon State University, Corvallis, Oregon. Feb 1990.

[Shachter 1991] Shachter, R. D., Andersen, S. K., and Szolovits, P. The Equivalence of Exact Methods for Probabilistic Inference on Belief Networks. *Artificial Intelligence.* May 1991.

Appendix A

Year One and Two Annual Technical Reports

ADVANCED DECISION SYSTEMS
1500 Plymouth Street, Mountain View, California 94043-1230 • (415) 960-7300 • FAX (415) 960-7500

TR-1258-01

Image Understanding Environment for DARPA Supported Research and Applications

Annual Technical Report
Draft

Prepared by:

Advanced Decision Systems:

Tod S. Levitt
Scott E. Johnston
Scott Barclay
John W. Dye

Georgia Institute of Technology:

Daryl T. Lawton

Period Covered: September 26, 1989 - September 25, 1990
Contract No. DACA76-89-C-0023
ADS Contract No. 1258

Prepared for:
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209-1155

U.S. Army Engineer Topographic Laboratories
Code: CEETL-RI
Fort Belvoir, Virginia 22060-5546

Table of Contents

Table of Figures

1.0 Vision Environment Report Overview

This report presents the functional specifications and top-level constructs of the core design of an image understanding (IU) application development environment. It also addresses system engineering issues in applying the environment to develop workstations specialized for terrain analysis and medical applications. The environment build has also been started this year.

In addition to designing and building this nearterm environment, Advanced Decision Systems (ADS) and Georgia Institute of Technology (GT) have actively participated in the IU community's design of a DARPA-ISTO sponsored, portable IU software environment. This environment is intended to facilitate the transfer of IU community technology into industrial, military, and commercial applications. GT and ADS headed the design committee on knowledge representation in the preliminary design effort from 5/90 through 9/90, and are currently a part of an independent design team in the continuing design effort.

The core environment presented in this report provides tools to leverage the development of IU applications and to facilitate transfer of IU and reasoning technology from its origins in research laboratories into IU applications. The core environment provides both a development platform and reusable components including a library of image processing and IU routines and data structures, and an integrated set of higher-level reasoning capabilities such as bayesian networks and logic engines. This layered software evnvironment concept is pictured in

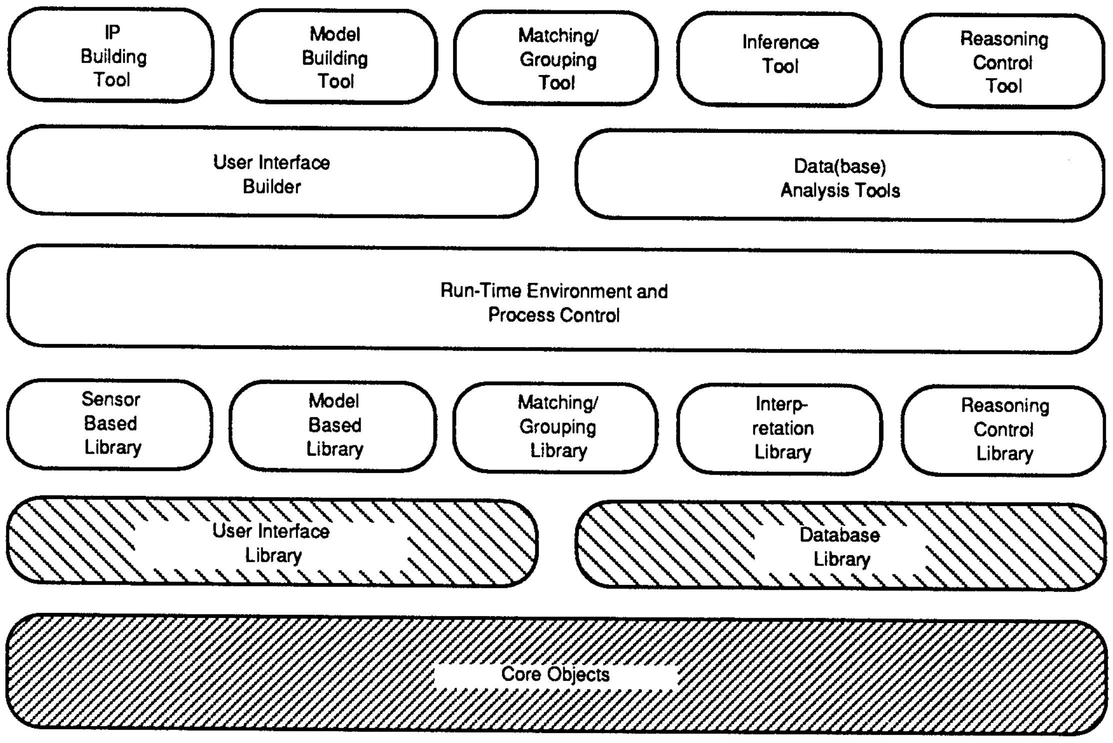


Figure 1: IU Application Development Environment Concept

This program is focused on the design and development of the core objects necessary for the foundataion of the environment. These correspond to the shaded boxes in Figure 1. The primary accomplishments of the project so far include

- Creation of a functional specification and top level design for an IU software environment integrating model-based reasoning, image processing, automated inference and hypermedia capability,

- Development of an IU environment class structure,

- Implementation of a partial object hierarchy, including build of a basic set of user interface and imagery manipulation classes, extension of image objects to include arbitrary gray level polygons, graphical interaction with remote databases, and Bayes net objects integrated with feature extraction capabilities,

- Providing a system engineering analysis of tasks and requirements for diverse IU applications and distilled a common core of IU workstation requirements, and

- Identifying and integrating key public domain software to provide image processing and user interface capabilities.

The environment design is an object oriented structure built on the C++ programming language. The design describes object representations that are used for the different classes of objects in the environment. Object representations are designed to provide a direct and useful interface to environment capabilities and programming constructs. The report also provides discussions of user interface, IU routines, inference, database and other capabilities, and how these facilities are integrated with the object representations.

The goal of the object oriented development approach is to build C++ objects to support interpretation of spatial and temporal data. This is primarily targeted for IU applications, though the techniques are applicable to other applications where reasoning about complex data is involved. C++ was chosen because it is a widely used, efficient, object-oriented extension of C, that facilitates the integration of public domain code, commercial programs and hardware devices.

The core set of C++ objects serves as a foundation for the representation of spatial, temporal and symbolic entities central to application development of IU and decision-aiding systems. It is intended that application developers will extend the object classes to create objects customized for their application.

The typical application IU system requires signal and/or image processing, symbolic reasoning, and inferencing capability, an interactive user interface for inspecting and manipulating the processing results, and an associated database for storing the original data and the derived results. Application developers require the following basic capabilities in their development environment:

- Objects that represent spatial/temporal data and models
- Objects that represent reasoning/inference knowledge
- Interactive display capability
- Remote storage and retrieval capability

The application developer can extend the baseline of objects and methods as required by their particular application. These objects promote the interoperability of higher-level C++ modules that conform to them. Figure 2 shows the relationships between these environment concepts.

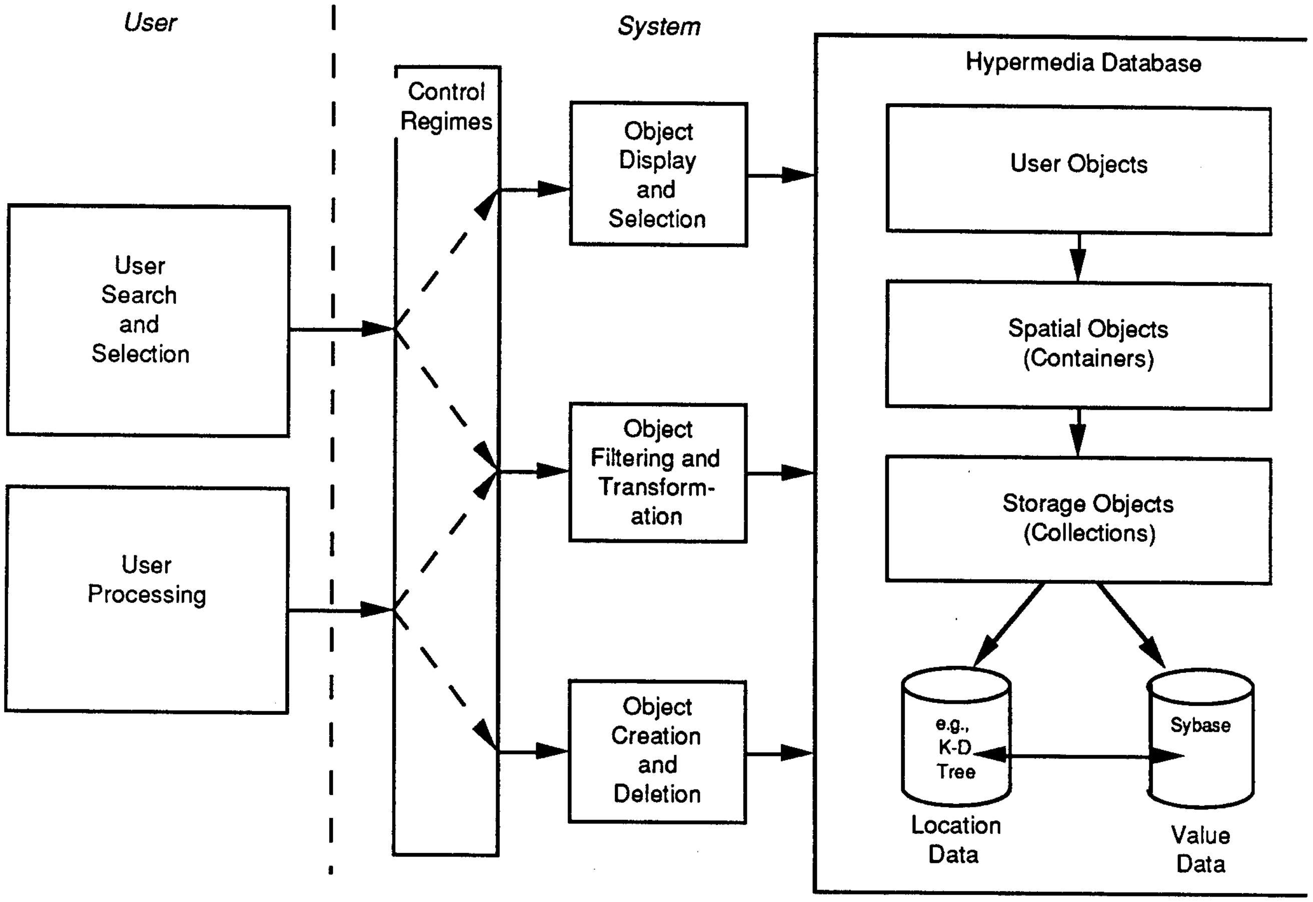

User
System
Control Regimes
Hypermedia Database
User Search and Selection
User Processing
Object Display and Selection
Object Filtering and Transform-ation
Object Creation and Deletion
User Objects
Spatial Objects (Containers)
Storage Objects (Collections)
e.g., K-D Tree
Sybase
Location Data
Value Data

Figure 2: Vision Environment Architecture

The remainder of this document presents our design of these C++ objects and their associated class hierarchy. The hardware and software environment assumptions are described in Section 2. Section 3 describes the core spatio-temporal object classes, section 4 describes user interface objects and methods, and section 5 discusses database classes. Together, these objects comprise the major functional components of the design. Section 6 shows the top-level set of code libraries. Code libraries can include objects, but much of a code library contains procedural routines for doing various tasks. These are mostly intended to be wrapped as methods for core objects, although it is possible for the application programmer to use them in traditional programming paradigms, and in novel uses such as functions for concatenation through mapping function wrappers (section 3.4.1.2). Section 7 presents key implementation issues that are considered in the design, including our approach to access of external databases, and the addition of code (sub-)libraries.

Terrain analysis and medical IU applications are presented in section 8. Terrain feature extraction from imagery and tissue segmentation in radiographs are analyzed to distill a common core of functionality required in the IU environment. This core provides the reusable infrastructure of the IU environment. Initial results are shown, including the re-implementation in the C++ environment of a Lisp/C program that performed model-based segmentation of hand radiographs using Bayesian inference for accrual of feature evidence.

Appendix A lists the core IU object class structure. Appendix B lists public domain image processing software packages that were considered for integration into the environment.

2.0 IU Environment Approach and Capabilities

Any IU environment aspires to support or possess all of the following goals and attributes.

- availability of algorithms
- execution efficiency
- interoperability
- verifiability
- portability

- extensibility
- coding efficiency
- function/data composability
- customizability

Efforts on other IU research and technology transfer environments [Quam, 84, KBVision, 87, Lawton and McConnell, 88, Lawton and Levitt, 89, Waltzman, 90] suggest that the first five goals are of primary importance for environments aimed at development of robust IU applications using well-understood IU technologies, i.e. technology transfer, while the second four are goals associated with rapid prototyping efforts common in IU research and innovative development of IU technology. This effort is focused at the goals that foster technology transfer.

In the following, we summarize hardware and software choices for a nearterm system build, then describe the fundamental philosophies and techniques for environmental software development. The third subsection presents results of programming instances designed to develop, test and demonstrate applications of the environmental constructs.

2.1 Workstation Hardware and Software Choices

Because of the bias towards technology transfer, and the desire to produce this environment within two years, environment component choices have largely been driven by current availability and prevalence of use of hardware and software options. Another driving factor was that as much as possible of the environment should be public domain, so that source code can be provided at minimal cost.

The basic development and user system is a Sun 3, 4 or Sparc workstation with a minimum of 12 megabytes core memory, keyboard and mouse, a color display and at least 300MB magnetic or read/write optical disk. A Vitek image processing acceleration board is under consideration for inclusion in the environment.

The software development environment is the Berkeley Unix 4.2 operating system on a Sun workstation, though compatibility to other Unix implementations and other workstations is maintained where reasonable. We have chosen C++ as the programming language. This choice is based largely on its efficiency, its relatively good compatiblity with C, and its nearterm widespread acceptance in the technology transfer community, i.e. the non-academic IU application development community. We have chosen the Free

Software Foundation's Gnu Compiler over AT&T's 2.0 C++ compiler for two reasons. The first is that the Gnu compiler generates faster, more efficient code because it is a true compiler and not just a preprocessor to a C compiler. The other reason is the availability of the compiler source code makes it portable to forseeable future (Unix) platforms.

X Windows is used for managing displays. The InterViews toolkit from Stanford provides a C++ interface to the X Windows package. IDraw, another Stanford product, provides the interactive graphic window interaction. Khorus, a public domain image processing library from the University of New Mexico, provides both the standard set of image processing functions as well as 2D plotting capabilities. Other public domain software packages being integrated in the basic environment include the CLIPS logic engine and rule-base package, the NCSA 3d display routines, and several neural net packages.

2.2 Core Spatial and Temporal Class Hierarchy

The class hierarchy is based on the structures developed in PowerVision and View [McConnell et. al., 88, Edelson et.al., 88]. In particular, the basic hierarchy of spatial classes and the concepts of transforms, function concatenation, virtual function wrappers, and programmable database-like search for perceptual grouping were all present in the original PowerVision implementation.

The current design has made strides in uniformity of these structures, cleaned up the relationship between objects and their display methods by associating display methods to the display objects (e.g. windows) rather than the source objects (e.g. a polygon), and has added class structures for coordinates. This design creates fundamental links between the geometric structure implied by coordinates and the programmability of search for perceptual grouping, as well as the linking together of lower dimensional spatial structures to form higher dimensional structures.

The core objects are organized into four general classes: scalars, collections, containers and coordinates. The scalars are the standard numerics and symbols of C++. Collections are general groupings of objects including arrays, streams, and graphs. Containers are groupings of objects that necessarily have an implied dimensionality and corresponding coordinate systems and imbedding spaces. Containers are inherently spatial: images, curves, solids, voxels, polygons, etc. Coordinates are objects that represent coordinate systems. Local coordinates are objects that are necessarily included within other objects (including other coordinates), while global coordinates can be disembodied.

Containers are designed to wrap around collections, and embed them in a coordinate system. Loosely speaking, we think of the semantic objects in IU systems, such as images, surfaces and volumes, as collections of values associated with coordinate systems. The grouping together in a systematic way of collections with coordinates forms containers. An array of integers is a collection. An array of integers associated with coordinates indicating the context of the array in pixels and centimeters is a container that is of the class Image. Figure 3 shows how containers, coordinates and collections relate to each other, and how they fit into an overall system.

Containers necessarily have coordinate objects and are closely tied to the user interface. The coordinate systems of the containers can map into the display coordinate systems. The display window itself is represented as a container. The necessary projections, translations, rotations, and scaling are implemented by "virtual" containers that wrap around previously instantiated containers and convert them into the appropriately appearing object.

Collections do not have associated coordinate objects, although they can have indices, such as indexes for an array. Collections are closely tied to the underlying devices. For example, a collection can be made to correspond to a device such as an image scanner. The scanned image becomes an array (one representation of a collection). Efficient access, traversal and transformations are built as methods on collections. Another example is a neighborhood operation like convolution. It can be realized as a collection of data and a method that manages buffers to create fast virtual memory access to the data in the collection.

Tranforms are procedures that operate on containers, coordinates and collections and produce containers, coordinates and collections as output. Although it is possible to represent transforms as containers, providing a pleasing uniformity of data types, it can be semantically confusing to the user. Because technology transfer is a fundamental goal, we have erred on the side of clarity rather than uniformity. So an image is called an image, for example, instead of a function that represents a 2d surface in 3space. We intend to overload class names to permit users both views of appropriate objects.

Where it is not confusing, transforms are represented as overloaded constructors of the class of their output objects. For example, a histogram is a constructor method for the one-dimensional signal that is the output of the histogram transform on an image.

When possible, transforms are defined on containers but implemented on the (coordinate-free) collections to maximize reusability. For example, a one-dimensional smoothing filter can be implemented on an array, then be usable on any linear collection of data, such as an image row, a curve in 3 space, or a specific traversal of the edges of a solid. So the filter can be represented at the more abstract level of the container hierarchy as a method on a curveNd (i.e. a one-dimensional curve in N space), enabling polymorphism.

In section three, collection, coordinate and container objects are described in detail. There is also a description of how containers and collection objects are efficiently traversed, accessed and searched.

2.3 Current Environment Capabilities

The current environment status represents five months of design and three months of implementation on a 27 month effort. The Khorus image processing package has just become available as of the writing of this report and is not yet integrated in the environment Therefore, implementation results focus on basic user interface and database capabilities.

InterViews and IDraw are public domain object oriented user interface toolkits built on top of X Windows. To date, ADS has extended the graphical object hierarchy of InterViews and IDraw in two ways: the addition of images and of Bayes nets.

Within IDraw, images are first class objects. The user can put an image object into the drawing by clicking with the mouse and pulling out a rubber rectangle to define the outline of the image. The system presents the user with a menu of files, and when the image file has been chosen, inserts the image into the designated rectangle in the drawing, clipping the image if necessary.

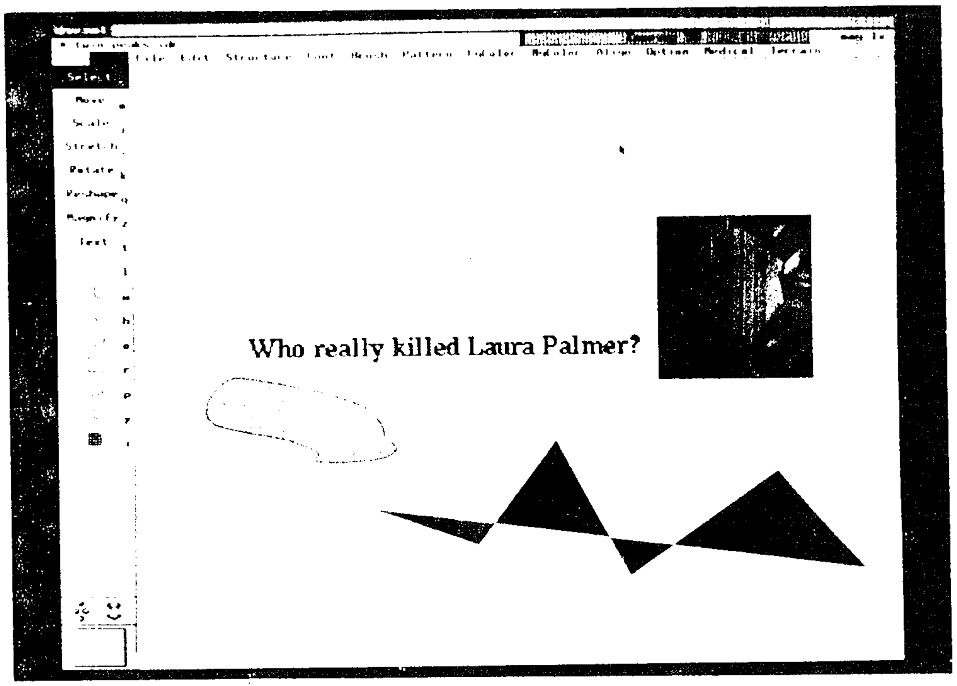
Once an image object is defined and displayed, the user can perform a variety of drawing operations on it, as with any drawing object. Images can be moved, scaled, stretched in width or height, or rotated. Arbitrary image warping is currently being implemented. In addition the user can draw any kind of graphical object on top of the image, and then group the object with the image, allowing drawing operations to be performed on both objects simultaneously. For example, the user can draw a colored polygon over a region of interest on an image, then group the polygon with the image into a composite object, then scale and rotate the composite object. The polygon still covers the same area of interest on the image. These capabilites are shown in figure 4. (To save space, some photos have been cropped so that the full computer screen is not shown.)

Rather than detailing each additional capability, we show our current state of implementation through two interactive processing scenarios. They demonstrate the benefits of an integrated object hierarchy, the use of images as first class objects, uniform representation of display and interactive object manipulation, and seamless access to remote processes.

The first application is diagnosis of arthritis from evidence extracted from a hand xray pictured in figure 5. Nodes and links are included as graphical objects. Graphically accessible methods are associated to form, in this example, a Bayes net object. Evidence can be acquired from images by measurement, and the evidence propagated through the Bayes net. Probabilities can be graphically inspected. For example, given a Bayes net that draws inferences about a disease condition of arthritic hands called periarticular demineralization, it is possible to take measurements on an xray of a hand in order to obtain evidence for the Bayesian network.

As illustrated in figure 5, the user loads the xray as an image object, draws a line down the middle of one of the finger bones (phalanges) and asks for a plot of the intensity values under the line by selecting "Profile" from a menu. The plot is shown in a window. A measure is taken of the relative density between the ends of the phalanx and the average density along the axis. This measure is added to the Bayes net as evidence by selecting the image, line and relevant Bayes node and selecting "Add Evidence" from the list of Bayes net menu options. The impact of the evidence at any point in the net can be seen by selecting the desired node and the menu choice "Show Belief". It is displayed as a probability histogram over the possible hypotheses at the node. In figure 5 these hypotheses are "demineralization" and "normal".

The second application scenario involves interactively querying a digital terrain database stored in Sybase. Figure 6 shows seamless interaction with an external process through a graphical interface. The user brings in an image of a map that is registered with the digital database. A region of interest is selected by drawing an ellipse on the map. Selecting the appropriate terrain layers and the "Retrieve" option from menus, a message is sent to Sybase, generating an SQL query. In this case, the database is populated with data on offshore oil wells, so a popup window of the wells in the region is displayed when the query results are returned.



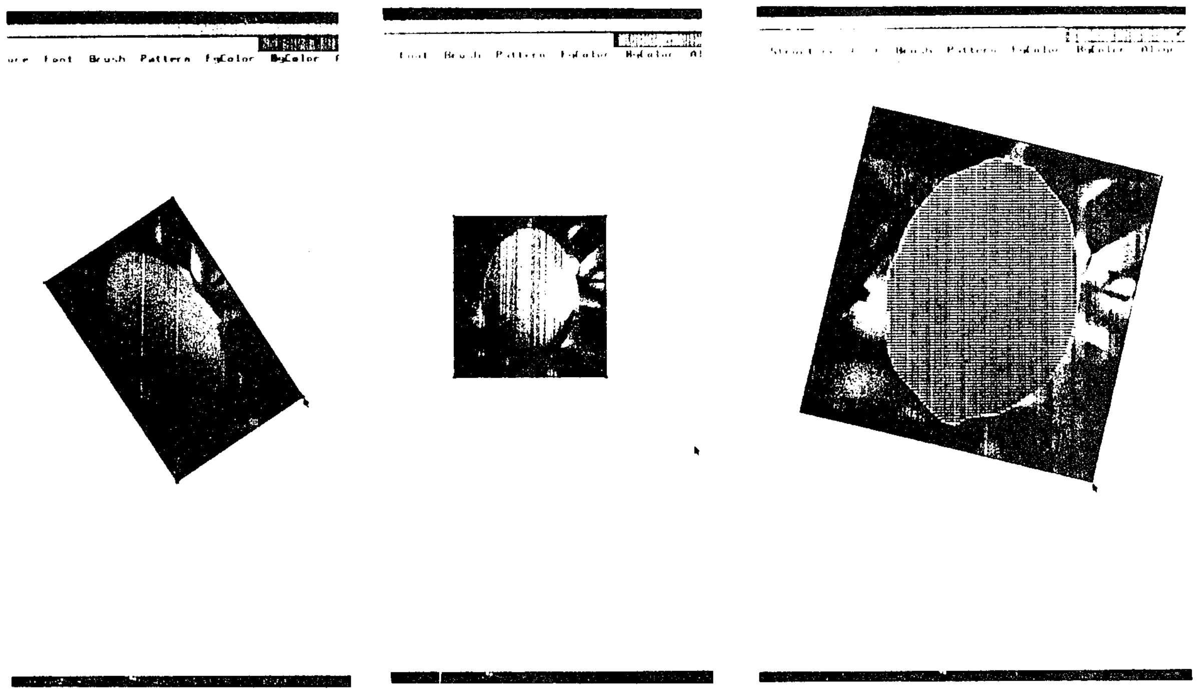
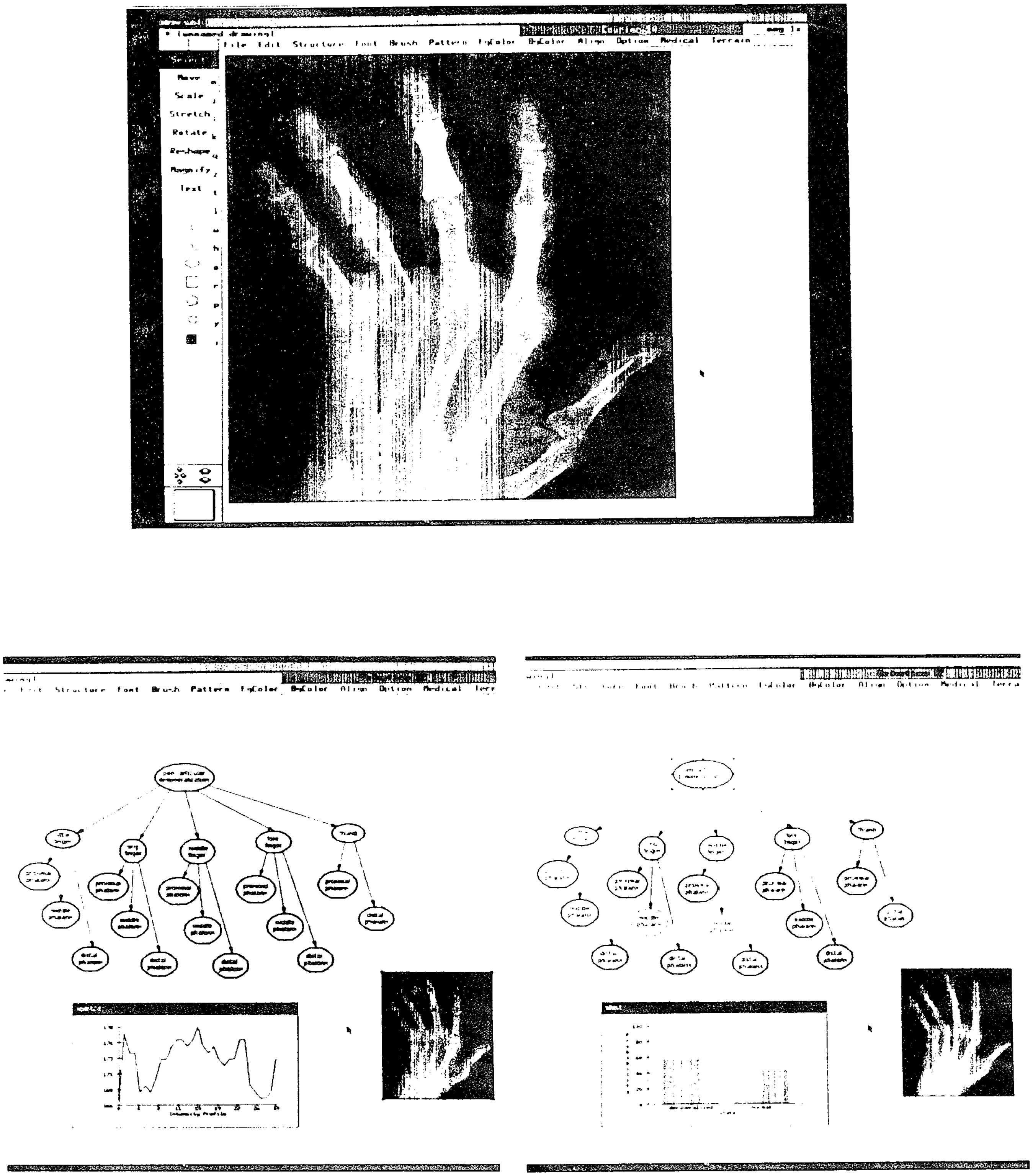
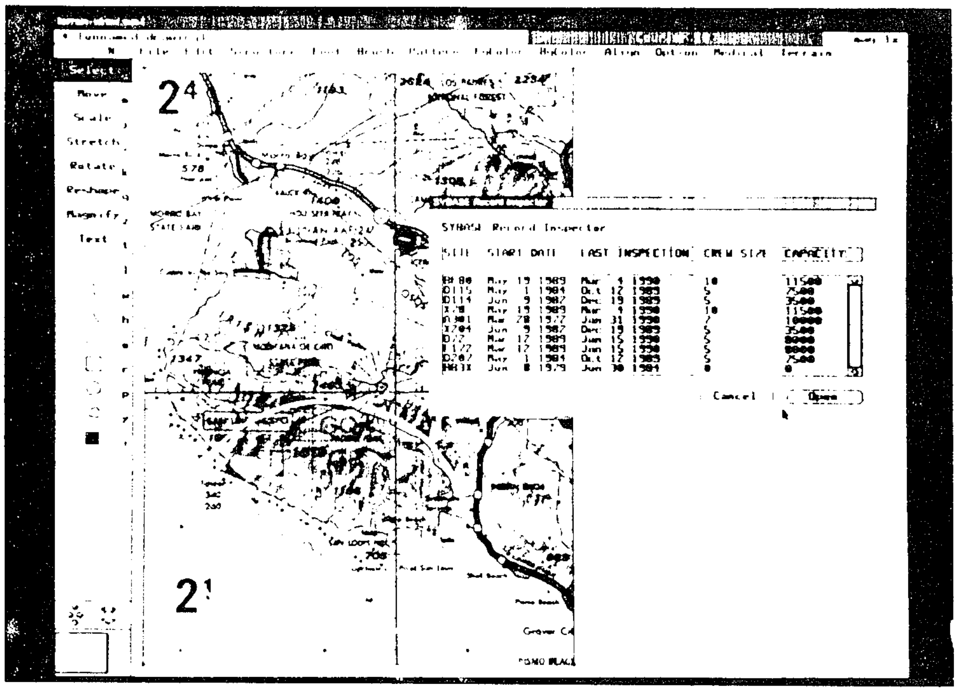


Figure 4. Image Display, Manipulation and Graphic Overlay



File Edit Structure Font Brush Pattern FgColor BgColor Align Option Medical Terrain
Select
Move
Scale
Stretch
Rotate
Reshape
Magnify
Text
thumb
Intensity Profile

Figure 5. Integrated Image Feature Extraction and Bayesian Inference



SYBASE Record Inspector

SITE	START DATE	LAST INSPECTION	CREW SIZE	CAPACITY
[illegible]	May 19 1989	Mar 4 1990	10	11500
D115	May 1 1984	Oct 12 1989	5	7500
D114	Jun 9 1987	Dec 19 1989	5	3500
[illegible]	May 19 1989	Mar 4 1990	10	11500
[illegible]	Mar 28 1977	Jun 31 1990	7	10000
I204	Jun 9 1987	Dec 19 1989	5	3500
D72	Mar 17 1989	Jun 15 1990	5	8000
[illegible]	Mar 12 1989	Jun 15 1990	5	8000
D207	May 1 1984	Oct 12 1989	5	7500
[illegible]	Jun 8 1979	Jun 30 1984	0	0

Figure 6. Graphical Sybase Interaction

3.0 Core Spatial and Temporal Objects

The core objects are high-level data structures in the technology domains of interest: image/signal interpretation, model-based reasoning, and hypermedia. The core objects are inherently hierarchical, with objects that decompose into member objects, that decompose into further objects, until the inner-most scalar objects, or values, are obtained. One of the advantages of the hierarchical representation is that it takes advantage of the (multiple) inheritance of attributes and methods that is built into C++.

The core objects are organized into four general classes: scalars, collections, containers and coordinates. The scalars are the standard numerics and symbols of C++. Collections are general groupings of objects: arrays, streams, graphs. Containers are groupings of objects that necessarily have an implied dimensionality and corresponding coordinate systems and imbedding spaces. Containers are inherently spatial: images, curves, solids, voxels, polygons, etc. Coordinates are objects that represent coordinate systems. Local coordinates are objects that are necessarily included within other objects (including other coordinates), while global coordinates can be disembodied.

Containers are designed to wrap around collections, and embed them in a coordinate system. Collections and coordinates are used in various ways to build containers. Figure 3 shows how containers, coordinates and collections relate to each other, and how they fit into an overall system.

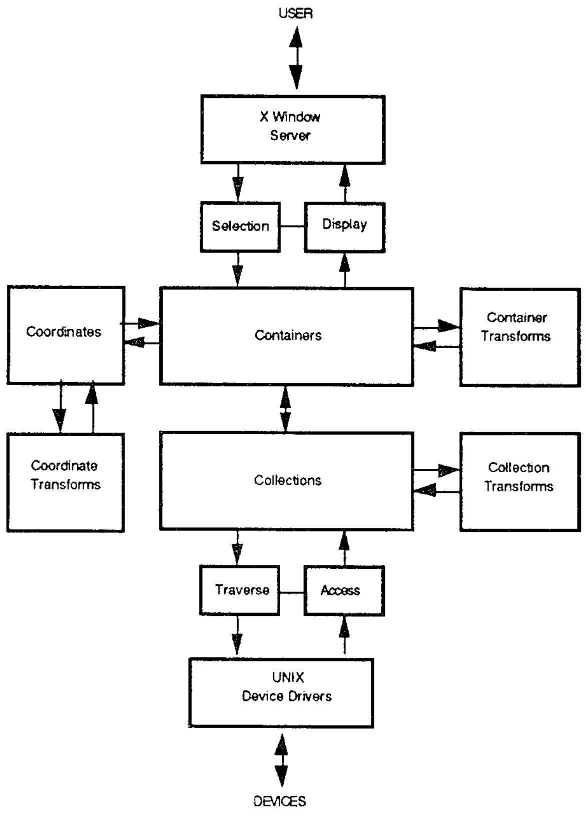


Figure 3: Core Object Relationships

Containers necessarily have coordinate objects and are closely tied to the user interface. The coordinate systems of the containers can map into the display coordinate systems. The display window itself can be represented as a container. The necessary projections, translations, rotations, and scaling are implemented by "virtual" containers that wrap around previously instantiated containers and convert them into the appropriately appearing object.

Collections do not have associated coordinate objects, although they can have indices, such as indexes for an array. Collections are closely tied to the underlying devices. For example, a collection can be made to correspond to a device such as an image scanner. The scanned image becomes an array (one representation of a collection). Efficient access, traversal and transformations are built as methods on collections. Another example is a neighborhood operation like convolution that can be realized as a collection of data and a method that manages buffers to create fast virtual memory access to the data in the collection.

Tranforms are procedures that operate on containers, coordinates and collections and produce containers, coordinates and collections as output. Where it is not confusing, transforms are represented as overloaded constructors of the class of their output objects. For example, a histogram is a cosntructor method for ValuedCurve1d that is the output of the histogram transform on an image. Where possible, transforms are defined on containers but implemented on the (coordinate-free) collections to maximize reusability. For example, a one-dimensional smoothing filter can be implemented on an array, then be usable on any linear collection of data, such as an image row, a curve in 3 space, or a specific traversal of the edges of a solid. So the filter can be represented at the more abstract level of the container hierarchy as a method on a curveNd (i.e. a 1 dimensional curve in N space), enabling polymorphism.

The next three subsections describe the collection, coordinate and container objects in detail. This is followed by a description of how containers and collection objects are efficiently traversed, accessed.and searched.

3.1 Collection Objects

Three classes of collection objects are planned: Stream, Graph, and Array. They can be characterized by the style of traversing and accessing the collection. Streams are traversed in a sequential manner, where the next access is restricted to the neighbor in a single forward direction. Graphs are traversed in a linked manner, where the next access is restricted to nearest neighbors in any direction. Arrays are traversed in a random manner, where the next access is unrestricted. IS-A hierarchy of collections in Figure 7.

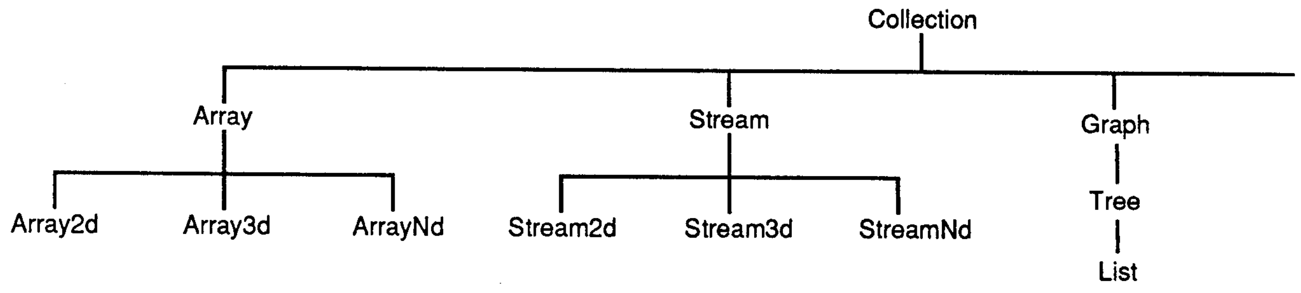


Figure 7: Collection Objects

3.1.1 Collection Classes

Any collection can group any set of core objects. Arrays can collect other arrays, or streams, or graphs, or scalar objects. Specific subclasses and/or methods are supplied for optimizing homogeneous sets of scalar objects.

The collection class hierarchy can be extended to wrap a stream, graph, or array around external sources. A character stream can be wrapped around a serial port. An array can be wrapped around a frame buffer. A graph can be wrapped around a Connection Machine or Transputer topology.

3.1.1.1 Streams

Streams are inherently linear collections. A series of objects is followed by an end-of-stream object. Higher dimensional streams are represented as nested streams. A buffered stream is supported to speed access of stream objects.

3.1.1.2 Graphs

Graphs are the general case of a linked data structure. A tree is a subclass of graph, and a list is a subclass of tree. This class hierarchy allows lists and trees to be manipulated by graph traversing routines, e.g.,"WalkDepthFirst". Graphs can store heterogeneous collections of objects. Graphs are implemented with separate node and arc objects. At this time we do not plan to support special subclasses for specific types

3.1.1.3 Arrays

Arrays are randomly-accessible object collections. The most general array is a linear collection of objects. Subclasses are defined that allow this linear collection of objects to be indexed with 2, 3, or N indices.

3.1.2 Collection Methods

The basic methods for a collection are as follows:

constructors	-creation and conversion routines
destructors	-memory/process/device deallocation routines
printers	-ASCII printing routines
traversers	-universal location generation routines
searchers	-selective location generation routines
accessors	-value access routines

"Constructors" allocate memory for a given object, then initialize this memory as required. This includes initializing the mechanisms for storage/retrieval of data to and from arbitrary sources (i.e. disk, display, digitizers, network, video disk).

"Destructors" deallocate memory, terminate, close and/or reset processes, devices and mechanisms as required.

"Printers" generate ASCII formatted representations of an object, both pretty-printed concise representations and full dumps.

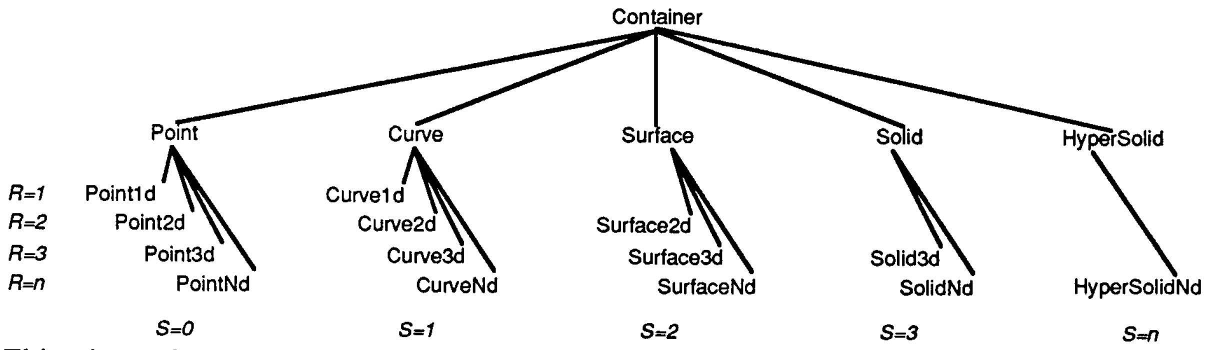
"Traversers" are methods for traversing the object, visiting each member object or element in turn. Each traversal of an object has an associated current location. Traversers accept a function pointer argument and apply the function at (a neighborhood of) each location.

"Searchers" are incremental, partial traversal methods. A search routine operates at the current location and chooses which location(s) to visit next., A search routine accepts two function pointers, applying one to the current location (neighborhood) and the other to choose the next location(s).

"Accessors" are methods for accessing the member object stored at the current location in the collection object. Access can be by value, or by reference to allow for overwriting.

3.2 Container Objects

Container objects represent an S-dimensional containment of objects in R-space.



This is only the top of the container class hierarchy. Additional subclasses are derived so that the leaf node classes of the hierarchy are realizations of more familiar spatio-temporal data structures and procedures. A signal is subclass of a one-dimensional container in 1-space. An image is a subclass of a two-dimensional container in 2-space. A volumetric representation is a three-dimensional container in 3-space. Constraints for a linear programming problem can be viewed as an N-dimensional container in M-space.

The container subclasses are effectively parameterized by the dimensionality of the container's topology, and the dimensions of the imbedding space. A 2d curve is a one-dimensional container in 2-space. A 3d curve is a one-dimensional container in 3-space. A polygonal region is a two-dimensional container in 2-space. A polygon3d is a two-dimensional container in 3-space. A terrain elevation map is a two-dimensional valued container in 2-space.

Specific classes of container are represented in multiple ways. For example, a three-dimensional container in 3-space is a solid, and a solid can be represented functionally (X^2 + Y^2 + Z^2 <= 1), volumetrically (via voxels or octtree), or by a surface model.

The cross-product of the S-dimensionality of the containers and the R-dimensionality of the embedding space is represented by a class hierarchy where the top-level branching is container dimensionality and the lower-level branching is embedded space dimensionality. Point, Curve, Surface, Solid, and HyperSolid are the superclasses, and their subclasses correspond to the space the container is in.

To achieve efficiency, 0d, 1d, 2d, and 3d containers are implemented as special-cases, and Nd containers are handled in a general fashion. In the same manner, containers embedded in 1d, 2d, and 3d spaces are implemented as special cases, and embedding in Nd is handled in a general fashion. Beneath each branch of the container hierarchy are three subclasses that reflect increasingly general ways of representing a container:

1- Constant Containers
2- Valued Containers
3- Connected Containers
4- Aggregate Containers

Constant containers describe the shape of a container without representing its values , or "contents". The shape is defined to be its geometric representation in Nspace, without values necessarily being defined at locations of the shape. Because a shape is geometric, it usually has a boundary that we call its "shape boundary" to distinguish it from other uses of the term. For example, a solid cylinder in 3space has a solid cylinder as its shape, and a hollow cylinder as its boundary shape.

We can represent a force field acting on the solid cylinder by associating the appropriate local magnitude and direction of the force field with each point of the shape. This is an instance of a valued container. Valued containers have a shape description and a content mechanism, whereby values such as scalars or more complex objects can be associated or stored with each shape location.

Connected containers group other containers, relating them with a series of coordinate transforms or other relations such as adjacency or attachment, and merging them into a single connected entity. A CAD model of a single car built from surface facets is a connected container. A smoothing pyramid is a connected container, where image objects are related (connected) by the order of the smoothing and sub-sampling operations that created them.

Aggregate containers group a disjoint set of containers into a single entity. The set of CAD models of all cars manufactured at a particular plant is an aggregate container.

3.2.1 Container Classes

3.2.1.1 Point

Point has four subclasses, Point1d, Point2d, Point3d, and PointNd. Each represents a single location, without length, area, or volume, in the particular space of that dimension. For example, a Point2d is a point in 2 space.

3.2.1.2 Curve

Curve has four subclasses: Curve1d, Curve2d, Curve3d, and CurveNd. Curve1d is a standard, one dimensional signal.

"Walk_locations" is the general traversing mechanism for curves. It walks down every pixel (voxel) that lie on the curve's paths. "Walk_vertices" and "walk_edges" are supported as well for curves that are represented as collections of vertices or edges.

3.2.1.3 Surface

Surface has three subclasses: Surface2d, Surface3d, and SurfaceNd. A one-dimensional surface is not necessary. Surface2d represents images as well as the 2d regions used in 2D modeling and 2D graphics.

The traversing mechanisms for surfaces (and solids) are grouped into two categories:

1- shape boundary traversal
2- shape traversal

Only the shape traversals are defined as methods on the surface object. The shape boundary traversals are available as methods of the boundary object (a curve) that surrounds the surface object. The methods concerned with traversing the shape of a surface are:

walk_locations	traverse internal points of surface
walk_vertices	traverse graph of surface regions where vertices are nodes
walk_edges	traverse graph of surface regions where edges are nodes
walk_faces	traverse graph of surface regions where faces are nodes
walk_segments	traverse series of straight-line segments that make up surface regions

Specialized traversals are supported based on a particular underlying implementation of a container. For example, a run-length encoded container has a method of traversing by run-lists.

3.2.1.4 Solid

Solid has one subclass, called Solid3d for conformity. Lower dimensional solids are not necessary. SolidNd describes shapes in higher dimensions that have only 3d volume, and no higher dimensional "mass". Solid3d supports volumetric models, surface models, and functional models of 3d shapes.

To traverse the boundary of a solid, the shape boundary container(s) is extracted, e.g., surface2d, and the traversing methods of that container are used. To traverse the shape of a solid, these methods are applied:

walk_locations	traverse internal points of solid
walk_vertices	traverse graph of solid regions where vertices (junctions) are nodes
walk_edges	traverse graph of solid regions where edges of solid regions are nodes
walk_faces	traverse graph of the faces of solid regions
walk_solids	traverse graph of solids regions
walk_segments	traverse series of straight-line segments that make up solid regions

3.2.1.5 HyperSolid

HyperSolid has one subclass, HyperSolidND. Lower dimensional hypersolids are unnecessary. These can be represented with generalized volumetric techniques, surface representation (collection of hyperplane constraints), and functional representations.

Traverses are the same as for solids, generalized for N-dimensions.

3.2.2 Container Methods

The basic set of methods for a container are the following:

constructors	-creation and conversion routines
destructors	-memory/process/device deallocation routines
printers	-ASCII printing routines
traversers	-universal location generation routines
searchers	-selective location generation routines
accessors	-value access routines
draw	-draw representation of self in an Xwindow
display	-add self to display list (see section 4.2)
inside	-predicate to determine if point is inside boundary

"Constructors" are a primary mechanism for deriving new objects from old objects. Constructors can be used to create aggregate objects by grouping other objects. Constructors can wrap up other objects, and add functionality to transform them into the new object. A particular class can have several different constructors, each identified by its unique argument list. The arguments of a constructor can be by reference or by value. When objects are passed by reference to a constructor, the newly created object incorporates the old objects into itself. Its functionality then depends on the state of its internal objects (or the objects it references).

"Destructors" deallocate memory associated with an object, and in turn invoke the "destructors" of their constituent objects.

"Printers" generate various formatted ASCII output of the container. This is similar to the collection "printers" with the addition of coordinate information.

"Traverser", "Searcher" and "Accessor" methods typically window through to an underlying collection. The current position of a container traversal is in effect a current position of the underlying collection traversal, and the mechanism for accessing the data in the container is the same as the mechanism for accessing data in the underlying container or collection.

"Display", "Draw" and "Inside" are methods for realizing the user interface.
See section 4 for details. The display method queues the object for display in an Xwindow by placing the object on the display list (see section 4.1) of the window. Then the window object takes care of determining the necessary parameters to call the object's draw method (see section 4.2). An object's draw method produces pixel values that are a representation of itself and maps them into a window display, or any container of type Surface2d. The inside method is used by the window object for each object on the display list to determine if a particular mouse click has fallen within its bounds (see section 4.4). Only containers and coordinates can be displayed in the 2D

and 3D object windows, as coordinates are required to relate to the window display. It is possible to display collections in structured text or graph browsing windows

3.2.3 Container Subclasses

The set of methods of each container class is the interface between objects of that class and the external environment. In a formal sense, the specification of these methods constitutes a contract with the external environment. The strong type definitions of C++ help ensure the correct form of all interactions with the object. When an object is passed as an argument to an arbitrary function, the set of methods define how the function can manipulate the object. When an older object is supplied as an argument to a newer object's constructor, the object methods operating on the older object present methods to the newer object that can then be exploited to realize the newer objects capability.

If the older object is passed to the newer object by reference, the two objects become related, and modifications to the older object are reflected in the functionality of the newer object. If the older object is passed by value (or the newer object makes an internal copy of the older object), then the two objects remain unrelated.

In general, the construction of containers involves the wrapping of a new container around a set of older containers. The class of the new container defines what it is. The classes recognized by the constructors define how it can be built. The information embedded in the container is a mapping from the containers supplied as arguments describing the functionality of the resultant new object.

The subclasses of the containers were designed with this in mind. The primary description of any subclass is twofold:

1- description of the capability of the container it implements
2- description of the set of containers (or collections) it can accept to construct this capability

The resultant subclasses are prefixed with Constant, Valued, Connected and Aggregate. Each subclass describes a basic approach to building containers. Constant builds the shape of a Constant container. Valued borrows the shape of an existing container and inserts new values.

Connected takes a set of containers and relations between them and groups them into a single container. The set of relations must be "path-connected" in the sense that given any two of the containers, A and B, there is a sequence of containers in the set starting with A and ending with B, such that there is a relation specified between any two containers that are adjacent in the sequence. For example, a CSG model of a tank that had coordinate transforms and attachments specified between adjacent primitive solids is a connected container.

Table 1 summarizes the possible ways of constructing these various containers from other containers and collections.

Table 1: Constructor Arguments

Container Subclass	Constructor Arguments
Constant Container	1) another container that represents the shape
	2) an existing collection and corresponding coordinate information
Valued Container	1) another container for the shape and an existing collection for the values
	2) another container for the shape and an existing container for the values
Connected Container	1) list of containers and relationships
Aggregate Container	1) list of containers

3.2.3.1 Constant Containers

Constant containers are descriptions of a region but do not describe the values associated with points contained in the region. Typically, constant containers are implemented as a representation of the shape of the container. These containers represent a locus in some space. The boundary of the container can be traversed, the insides of the container can be traversed, but no values can be retrieved. Constant containers can be constructed from a lower-dimensional container that describes its boundaries. Two-dimensional surfaces are bounded by a closed two-dimensional curve. Three-dimensional solids are bounded by a three-dimensional surface (which is in turn bounded by a three-dimensional curve).

3.2.3.2 Valued Containers

Valued containers describe the region as well as provide a method for accessing the values associated the region. The boundaries and insides of these containers can be traversed, as with Constant containers, and values can be extracted at each location in the traversal.

In general, valued, or "full", containers can be constructed by combining a Constant container object with a collection that maps locations in the container to values. This collection (an array, stream, or graph) can represent values inside the new container, or be restricted to locations on the boundary.

This general mechanism can be overridden by specific full containers that rely on a technique that entangles the boundary representation with the value representation. In this case their constructors do not have other containers (or collections) passed to them by reference.

3.2.3.3 Connected Containers

Connected containers group other containers into a single entity. The generalized composing mechanism relates objects by chaining them with a series of relationships, such as coordinate transforms, in the appropriate space. 3d objects can be chained with 3d rotations, scalings, and translations. 2d objects can be chained with 2d rotations, scalings, and translations. Other grouping methods, such as symbolic groupings of INSIDE-OF and ADJACENT are realized with subclasses of the general composition class.

More explicitly, the set of relations passed to the connected container constructor must be "path-connected" in the sense that given any two of the containers, A and B, there is a sequence of containers in the set starting with A and ending with B, such that there is a relation specified between any two containers that are adjacent in the sequence. For example, a CSG model of a tank that had coordinate transforms and attachments specified between adjacent primitive solids is a connected container.

The general composition mechanism can also be used to transform the local coordinate systems of an existing container, by grouping it with a constant container associated with a different coordinate system. Projecting a container into a coordinate system with less dimensions is not handled by the connected container mechanism, but is instead supported by the constructors of objects in those lesser dimensions.

Constructors for connected containers take a list of older (sub) containers and associated local-coordinate systems (implemented as coordinate objects, see section 3.3), and other relationships, such as attachment and adjacency, as by-reference arguments. This list can include simpler containers in the same space, i.e. a connected two-dimensional surface can accept two-dimensional curves, because they are degenerate cases of two-dimensional surfaces.

3.3 Coordinate Objects

Coordinate objects represent coordinate systems. A coordinate has a corresponding type that is one of cartesian, polar, cylindrical, spherical, quaternionic , or shape. Shape means the coordinate system is defined in terms of distinguished points in a container, like attachment points, or the ends of axes of sub-objects. A local coordinate is necessarily contained in another object such as a container or another coordinate. A disembodied coordinate is defined to be the subclass of global coordinate. Coordinates have methods that act as transformations between other coordinate systems. A coordinate records its transformations between other coordinates, unless these transformations are explicitly deallocated.

3.3.1 Coordinate Classes

There are two subclasses: global and local. Local has a special subclass called base-coordinate.

3.3.1.1 Global Coordinate

Global coordinates can occur disembodied, i.e. without being contained in or referencing other objects. They can be transformed and copied by any coordinate constructor to mix in when constructing a local-coordinate defined for a container or other global or local coordinate. This "places" the container in the global coordinate system. The global coordinate remembers the containers that were constructed with it.

3.3.1.2 Local Coordinate

A local-coordinate can represent the imbedding space of the container, or other ego-centered coordinate systems. An object can have multiple local coordinates. Each local coordinate that is constructed must have a transformation that represents it in the coordinate system defined by the base coordinate.

3.3.1.2.1 Base Coordinate

The base coordinate is a distinguished local coordinate. It is defined to be the first local-coordinate associated to a container or other coordinate. The base coordinate is instantiated by the container contructor. It can be specified by the caller of the contructor method. The base coordinate is guaranteed to have transforms associated to all other local-coordinates of that container or coordinate. It is intended, although not required, that the base-coordinate correspond to the natural traversal of an associated collection of values. For example, a raster image is a Surface2d container whose values are in an Array2d (collection). Its natural base coordinate is the cartesian coordinate with origin at array index (0,0), one axis in the row direction and another corresponding to columns. For a pyramid, a natural base coordinate is similar, but includes a third axis in the multi-resolution direction.

3.3.2 Coordinate Methods

Coordinates all contain the following methods. Note that when local and global are not explicitly called out, either applies. For example, the transform method can relate locals to locals, locals to globals or globals to globals.

type	- returns a mathematical type (e.g. cartesian) or the type "shape".
origin	- returns a point
dimensions	- returns list of dimensions
units	- returns list of named units per dimension
minextents	- returns list of minimum extents per dimension
maxextents	- returns list of maximum extents per dimension
convert	- inputs a type that is not "shape" and creates versions of its local-coordinates expressed in that type (e.g. cartesian to polar conversion)
list-transforms	- returns the list of transforms known between itself and other coordinates
transform	- inputs another coordinate with a known transform to itself, and a third coordinate with a known transform between it and the second coordinate; returns a transform between itself and the third coordinate.
propagate-transform	- inputs a coordinate with known transform between itself and the coordinate, and returns the list of transforms between all its local-coordinates and the input coordinate.

3.4 Object Traversal and Search

From the user's point of view, containers get traversed or searched. From the workstation environment's point of view, containers are pointers to collections that get traversed or searched. Both traversal and search can be thought of as routines consisting of the cyclic applications of three functions: move, access.and apply. In the case of traversal every value in the underlying collections is necessarily visited, so the function for moving, or choosing the next location(s) in a container's shape, is known before traversal is invoked.

In search all locations/values are not necessarily visited. The move function must be passed by reference to the search method. The apply function is invoked on the appropriate neighborhood at each visited location for both traversal and search methods.

Signal and image processing functions traverse their contents to enable extraction of higher-level interpretations. These routines need to quickly iterate across their N-dimensional data sources, with efficient access to a local neighborhood ranging in size from 1 to M units in any dimension.

Traversal is intended to provide the support for a programming style whereby the application developer codes the operation to be done at each point in the traversal, and leave it up to some other mechanism to slide this operation around the container. This requires two things: an underlying mechanism for efficient traversing (tied to an efficient accessing scheme) and a programmer interface.

Examples of underlying mechanisms are the tiling of imagery used in ADRIES, and the sliding-window subsystem from the Honeywell Image Research Laboratory. Each makes neighborhoods of pixels available to the programmer in an efficient manner.

3.4.1 Programming Traversal and Search

When a programmer is presented with an efficient source of neighborhood data, it is convenient to string together a series of smaller functions to do the work of a more complex function. However, the programmer is typically forced to write the complex function out flat, inline in one function, to avoid the overhead of piping data between functions. The IU workstation environment provides support to concatenate existing low-level operations without incurring extra overhead.

This can be done by constructing a library of neighborhood operations that describe what is done on one neighborhood, but contain no mechanism for iteration. Examples are convolution kernels, median filtering, and basic arithmetic and logical manipulations of Nd data. Neighborhood operations that maintain a state are implemented in this model by saving the permanent state in static and/or global variables for later retrieval.

This makes the process of writing more complex neighborhood operations into one of concatenating the series of operations into a single neighborhood operation. A specific neighborhood function is then inserted in the middle of a looping mechanism that is capable of traversing the container, and supplying the neighborhoods of data to the operator.

There is a split here that needs to remain clear. The reusable neighborhood operations are small code chunks that have no built-in looping mechanisms. An application specific neighborhood operation takes a sequence of reusable neighborhood operations, and wraps them up with a traversal or search mechanism. To reuse that specific neighborhood operation means the body of the loop has to be extracted and made into a stand-alone function. That effort is only desirable when the function is to be reused; otherwise it leads to extra overhead from a layer of function call. An example of convolution code is shown below. Convolution has a neighborhood application function, and an outer loop that traverses a container.

```
int ByteStream2d::convolve
(
   ByteStream2d *outstream;  // Output 2d stream of bytes
   ByteArray2d *mask;        // Convolution mask.
)

{

// Local variables
   int mask_width = mask->width();
   int mask_height = mask->height();
   int dummy;

// Ensure input and output streams are rewound
   this->rewind();
   outstream->rewind();

// Create and fill neighborhood cache on input stream
// Boundary handling is done by the cache.
   char **instream_cache =
      this->cache( mask_width, mask_height );

/* Load entire mask into its own cache */
   float **mask_cache =
      mask->cache( mask_width, mask_height );

// Loop until double end-of-stream
   while( !instream->eos() )
   {

   // Loop until single end-of-stream (end of row)
      while( !this->eos() )
      {
         int out_value = 0;

      // Convolve mask with neighborhood
      // and write result to output stream
         for( int i=0; i<mask_height; i++ )
            for( int j=0; j<mask_width; j++ )
               out_value += *(*(mask_cache+j)+i) *
                            *(*(instream_cache+j)+i);
         outstream->next() = out_value;
         dummy = this->next();
      }

   // Set up for next row
      dummy = this->next();
   }

// Return OK status
   return 0;
}
```

3.4.1.1 Neighborhood Cacheing

Object access is streamlined with a cacheing mechanism that makes a local neighborhood available to the C++ program in an internal C++ data structure. The mechanism is program-controlled, in that the program decides when to initialize it and when to refresh its contents. Because the cache is represented as a standard C++ data structure, either an array or a nested array of pointers to arrays, the efficiency of data access within the cache is identical to array-based data access.

A neighborhood cache is useful for representing windowing operations on imagery. The input object is an image, the cache is an array of pointers to linear arrays; as the window slides across the image, the cache is refreshed by updating the pointers.

For point transformations the cacheing mechanism is useful in order to reduce the overhead of row access. The cache is defined to be a single array equal in length to the image row, and it is refreshed after each row is processed. The processing of the row is done with a tight for-loop, with entirely in-memory data access.

The convolution example above illustrates the use of neighborhood cacheing for arbitrary convolution of imagery. Two caches are employed, one on the input image stream, another on the 2D-array that defines the convolution mask.

3.4.1.2 Mapping Function Wrapper

The ability to compose functions without creating intermediate data structures yields the ability to display the results of experiments with a minimum of typing on the part of the programmer (e.g. not creating named functions as above) and with a great saving of memory and memory management overhead. In Powervision (the ADS vision development environment built in ZetaLisp on a Symbolics lisp machine), functions created this way were called "pixel-mapping-functions". Because individual objects know how to display themselves, the pixel-mapping-function capability is re-created with a wrapper that says: compose the following functions (passed by reference) on this input data, and add the display method for the result of the last function at the end. Ordinarily, the final result is saved, because it is often the input to a next stage of processing.

For example, to apply a convolution to only the pixels in an image defined by a mask, a search method is applied to the image object, where the search method run length encodes the mask and accesses the "on" pixels only. The search method is composed with the convolution to feed only the relevant neighborhoods to the convolution kernel.

Mapping functions are implemented by subclasses of their respective containers. There are four basic types of mapping functions, and so four basic mapping function class extensions:

1) coordinate transformation of container locations
2) look-up-table applied to container values
3) arbitrary expression applied to container values
4) arbitrary expression applied to container locations.

3.4.2 Spatial Data File System

Two capabilities have been defined to support the traversal and access of spatial data stored in containers: 1) an efficient file access mechanism, for retrieving data stored on arbitrary devices, and 2) efficient indexing schemes for quickly locating data stored within a container. In combination these are called the Spatial Data File System.

Collection objects are supported on diverse sources of data: disks, frame buffers, digitizers, optical disks, even virtual memory. Constructors set up and initialize access to these devices, and subsequent use of the object's methods so that the underlying access mechanism is transperant to the application developer. Destructors close and/or reset device access as needed.

Device drivers simplify this transparency, by making all objects appear as a stream of characters (or a buffered stream of characters). Furthermore, the model of a disk device driver suffices for a large subset of devices that can be viewed as a contiguous collection of characters coupled with a random seek mechanism.

The spatial data file system supports the I/O of spatial data to disk-like devices. The spatial data file system differs from a normal Unix file-system in that it attempts to optimize the access of large collections of 2d, 3d, or Nd data.

The ability to efficiently index into specific data stored within a larger collection is required. Database indexing uses (multidimensional) trees (B-tree, kd-tree, quadtree, octree, etc.) or hashing to isolate a particular item in a list.

For example, a collection of curves written to disk can be implemented in the following manner:

1- an array of bytes on disk is defined as the low level object

2- a tree index is computed that maps from the curve i.d. to the byte offset within the file

3- an array of curves is defined that combines the array of bytes with the curve-to-byte tree, and results in an array of efficiently accessible variable length curves that (just happen to) reside on disk

4.0 User Interface

The user-interface supports the direct manipulation and inspection of all entities in the vision environment through a windows-menu-and-mouse interface. The user interface is based on X Windows, a network window system, and InterViews, a C++ package that defines basic X Windows objects. The user interface must be capable of displaying a list of containers to an X window, and mapping mouse clicks to specific objects in the display list. The following subsections present the display list object, window types and addresses issues in imagery display and mouse protocol.

4.1 The Display List

The display list is an object that keeps track of what set of objects is currently being displayed in a window. There is a single display list associated with each window. The display list is implemented as a connected container of 2d surfaces. The connected container groups two-dimensional points, curves, and surfaces, interrelating them with coordinate transforms and other programmed relations. Each container stored in the display list knows how to redisplay itself, and knows how to determine if a given point is inside or outside its 2d shape boundary or whether a given rectangle overlaps its shape boundary.

4.2 Windows

The window system supports overlapping windows, as well as neatly tiled windows, useful for applications once they reach a certain level of maturity.
While in overlapping mode, each window can be resized, repositioned on the screen, collapsed down to an icon, and expanded back to its original size and shape. When overlapping mode is disabled (tiling mode), the size and shape of each window is predetermined (or tightly controlled), and the opening and closing of windows is directed by the application. Each window is associated with a display type that governs the type of information that can be shown within the display region.

The supported window types include:

1. **2D Object Display Window**-- images, lines, curves, 2D graphics

2. **3D Object Display Window**-- volumes, surfaces, 3D graphics

3. **2D Plotting Window** -- 2 axis: signal plotting, time series analysis, statistics, measurement or feature spaces in two dimensions

4. **3D Plotting Window**-- 3 axis: measurement or feature spaces in three dimensions

5. **Directed Graph Browser Window**

6. **Structured Text Browser Window**

7. **Dialog Box Window**

The display of objects to windows is object-oriented, in that each entity within the vision environment knows how to display (or present) itself to a window of a specific type. The following subsections discuss each window type.

4.2.1 2D and 3D Object Display Windows

An object display window presents a collection of image-understanding objects to the user, all registered to a single coordinate system with the display based on a single viewing perspective in that coordinate system.

The coordinate system of a 2D object display is that of the base coordinate of the primary image associated with the display. The typical viewing perspective of a 2D object is such that the primary image is displayed at full resolution. If the window size exceeds the primary image size, the remainder is filled.with a constant value If the primary image size exceeds the window size, the image is clipped. From this starting point, the viewing perspective can be adjusted to arbitrarily zoom and scroll the display.

3D display involves projecting 3d objects onto a 2d space, then entering the resultant container into the display list. This projection is done by the constructor routines of the 2d classes. For example, a constructor for 2d curves is supported that accepts a 3d curve and a set of coordinate transforms that describe the relationship of the 3d object's coordinates to the screen of the desired 2d projection. The resultant 2d container can transform surface orientation into intensity values of 2d surfaces. Hidden surfaces are dealt with by this projection mechanism as well.

The coordinate system of a 3D object is centered around the origin of some primary object associated with the display. The viewing perspective is based on the unit screen at a unit distance from the viewers focal point. A default 3D viewing perspective can be defined as a global coordinate object and placed at a distance from the viewer where the bounding rectangle of the unit screen matches the bounding rectangle of the object (padded with constant values to maintain aspect ratio) projected onto it (i.e, it fills the screen). From this starting point, the unit screen can be translated and rotated in 3-space to a new position that results in a new projection of the 3D object onto the unit screen, and a new display. Fant's warp and perspective algorithm is used for this method.

Each object in a 2D display is z-buffered to achieve an ordering from front to back, allowing for the overlay of graphical objects on top of other graphical objects (or images). This ordering also resolves which object responds when the cursor is positioned on it and the mouse is clicked. Ordering in 3-space is dependent on the viewing perspective, but for essentially 2D objects that lie on a 3D surface there can be a concept of ordering with respect to a particular side of that surface.

Any object that can display itself with the proper dimensions can be added to the list of objects displayed by a window. The following classes are supported for 2D: grey-scale images, binary images, labeled images, lines, curves, polygons, and any fixed projection of a 3D object to 2D space. Both volumetric and surface models are supported for 3D object display. Hidden line removal and other surface rendering techniques are supported as needed.

Color is required for drawing secondary objects on the face of primary objects, such as lines and curves embedded in an image, or the vertices of a 3D object that lie on its surface.

There is a need for "snap-to" capability that associates the position of the cursor when a mouse-button is clicked to the nearest reasonable object. This aids the user in selecting objects of single pixel width.

4.2.2 Plotting Windows

Plotting windows share traits with object display windows. The main difference is that they graphically present information that is inherently numeric, whereas object displays graphically present information that is inherently pictoral. The extent of each numeric dimension is presented to the user as an annotated axis, divided by tic marks into numeric ranges. By, default the plot is scaled to fit within the current window size.

2D plots consist of line plots, scatter plots (by dot and by character), and bar plots. Grids to aid in viewing are optional. 3D plots consist of surface grid plots (with hidden lines removed) or chunky bar plots.

In a similar fashion to object displays, any object that can present itself to the plotting window as a collection of numbers of the proper dimension can be added to the list of objects plotted by the window. The default ordering is FIFO, but can be modified by the user.

4.2.3 Directed Graph Browser Window

Arbitrary directed graphs, trees, and lists can be used within the environment to group together objects. The directed graph browser is provided as a general tool for graphically inspecting these data structures, and editing their contents. In general, the directed graph browser supports an arbitrary directed graph, but works just as well on the simpler data structures of trees and lists. It automatically positions the nodes within the window, drawing arrowed lines to show the relationship with other nodes, and asks each node to display itself, either by icon or by name.

Navigation of the graph can be done by the user or under program control. An advanced browsing feature is a miniature map of the entire graph, used to navigate when the graph is too large to fit in the window.

Nodes in the graph can be selected by clicking left. A menu of possible node operations is brought up by clicking middle, once the node has been selected. Certain node operations require a target node that is then selected by the right button after the first two operations.

The directed graph browser can be set up for read-only access of the directed graph, or with read-write permission can be used for interactive editing of the structure. Nodes can be created, deleted, and moved. In a similar fashion entire sub-graphs can be created, deleted, and moved. The structure of created sub-graphs is based on a list of default structures, or controlled by the program. Hierarchical graphs (nested) may be supported as well.

4.2.4 Structured Text Browser and Dialog Box Windows

The input/output of text and numbers is done through structured text browsers. At its simplest (initial capability), this is a text editor. At its most complex (eventual capability), it is a text I/O window that automatically enforces a certain grammar, such as the legal relationships that can be entered into a semantic network, the syntax of a programming language, or a database table structure.

Once again, as in the directed graph browser, text display in this window can be either read-only or read-write. A read-only text display, augmented by certain control buttons, is a dialog box.

Actions can be associated with the modification and/or selection of any text within the window. By clicking on a word in a list of words, it is possible to bring up another window with information pertaining to that item. This rudimentary hypertext is useful for following a chain of related objects in the vision environment.

The user interface is an easy to use hyper-text or hyper-media interface, built around the data constructs necessary for IU: 1, 2, and 3D signals and structures, and diverse networks and databases to contain them. To support IU further, links between various displayed entities are arbitrarily complex transformations, instead of simple pointers, to make for a flexible approach to prototyping applications. A single plane in a 3D plot can be selected, and displayed in a 2D plot window. If the new window is created by reference (versus by value, to borrow programming language terms), a modification in the original 3D plot is passed on to the 2D plot.

4.3 Overlays and Sprite Objects

Most graphics that overlay images occupy a small area compared to the image size. An example is the overlay of linear features, such as roads or rivers, on an image. When the image is drawn, it is from one container object. Each window is associated with a display type that governs the type of information that can be shown within the display region. The linear features are assumed to be a second container object with coordinates that overlap the image. The problem is to allow the user to select and unselect the display of the graphic overlays without redrawing the whole screen just to refresh the small area under the graphic overlays. The approach is to create a third object that has the same container (i.e. "shape") information as the graphics, but uses the values from the image collection. Refreshing the screen is accomplished by requiring the appropriate set of graphic objects to refresh themselves. This capability is called a "sprite" object in the object-oriented imagery display literature.

4.4 Visual Pointer

The initial pointer device is a three button mouse, because of the choice of Sun as the initial hardware platform. A mouse is used to manipulate a cursor on the screen. Action is taken when a particular mouse button is pressed. The action taken is a function of which button is pressed, what window the cursor resides in (the top-most window if there is overlap), and where in the window it resides. Standards are being incrementally developed to govern the types of actions associated with each mouse button. The cursor may change in size or shape to indicate change of state of any particular application.

Each window comprises several mouse-sensitive areas within it. These areas may or may not correspond to obvious graphical clues, such as a menu item, button, or graphical object. For the most part the mouse-sensitive areas lie within the bounding rectangle of the window, with the single exception of pop-up menus (or detachable menus) that are attached to the borders of the window.

Mouse selection is implemented by capturing a mouse click, and interrogating each object in the display list in turn, to determine if the click fell upon it. The effective area of the click is expanded to some minimal resolution (e.g., 8 by 8 pixels) prior to perusing the display list. A mechanism is provided for determining all the objects that overlap this expanded area.

4.5 Image Scrolling

A key problem is displaying large images or maps; much larger than one display screen can show at a time. In addition, the user wants to be able to scroll around on this image or map, and to zoom in to selected sections of the image or map.

The basic approach is to divide up the overall image into rectangles that can be moved from disk to memory and from one place in memory to another in a very short time (considerable less than a second). These image objects are a subclass of the image class, called tiles, and have the smaller container class instances that correspond to each of their parts. A slight overlap in tiles may be allowed to resolve border problems. When the user elects to scroll the image, i.e. move mouse, the pointer position is used to select the additional image tiles needed and to recover them from disk. Latency is addressed by mating the default tile size to the minimum block size for retreival and the bandwidth of the cpu to disk (or other storage) channel. Note that it is assumed that the environment can find out the size of an image in external storage. In that case, it can be tiled as it is read in and stored in the local database for use during the working session.

4.6 Menus

Most windows in the vision environment have a region where fixed menu selections are advertised, plus a region that displays the variable information displayed by this type of window. Within the display region, the menus are a function of what object is selected.

5.0 Databases

A database provides for the management of many types of persistent data including the objects being operated on, the functions in the function library and the versions of each of these items. Management means keeping track of the items as well as storing or saving them from one execution of the program to the next (persistence). A single database (consisting of multiple files) is associated with each workstation. Multiple workstations in a cooperative environment require either a shared database or a distributed database approach (probably the latter).

This database accomplishes many different functions including:

1- simple persistent storage of images, objects, functions and other data,

2- flexible conditional queries to retrieve or compute instances of objects,

3- structural ordering of the object instances to provide fast, efficient access to the objects,

4- a consistent convention for accessing a variety of different objects with a minimum of coding effort, and

5- extensibility to support group data sharing on different physical databases.

The database manager is organized in a client-server model and consists of two components:

1- The database interface (or client) provides the interface to the database from any other programs. This interface is provided as methods that the other objects may invoke. Such methods include: insert, delete, save, find, etc.

2- The database server manages the storage and access to items assigned to the database including the allocation and deallocation of space. This includes creation, documentation, modification, access, and deletion of user objects (images, features, etc.) and programming objects (functions, documentation, numbers, characters, etc.).

Databases traditionally provide a shared access that prevents two users from interfering with each other when accessing the same object and provide a transaction system to ensure the integrity of each interaction with the database. These aspects of a database are not as important for the IU environment and are not discussed further here.

5.1 General Database Approach

The approach to the database design is in two levels. These are:

1- The basic level provides for the storage of objects, groups of objects and indices as files for management by the operating system. These files can be stored and read directly from the program or under interactive user control.

2- The next level provides interface to database management systems from commercial products. The current plan is to develop a generic SQL interface for the class hierarchy. This allows interaction with standard relational database system (e.g. Sybase, Ingress, Oracle). Interface to or new object oriented databases (Ontologics' Ontos) is a future possibility. Hooks are provided to build additional structures for efficient access, such as quadtrees.

The database is integrated into the core workstation software in several ways:

1- The primary access to the database is through the C++ language. The user (developer) takes advantage of the database through the core object structure.

2- Our approach to the database uses the strong typing feature of C++ by allowing the database objects to be incorporated directly in the object hierarchy transparently. That is, the objects are compiled directly into the program (with strong type checking) and are stored in the database by invoking the persistence attribute.

3- All imagery, imagery objects, functions, production rules, reasoning structures, etc. are expected to be stored in the database and access through the same C++ program interface

4- A set of user interface procedures that form a front end to the objects stored in the database.are provided for the developer

5.2 Database Implementation

The Sybase relational database system provides the basic relational database capability. An object oriented view of the database is provided by a set of object oriented procedures used as a front end to the relational database and the objects are stored using some object oriented structures build on top of Sybase.

5.3 Object Storage and Retrieval

Objects that are capable of being stored in the database are given the attribute persistence (inherited from a high level object). When this attribute is turned on, the object is guaranteed to be stored in the database for later access. It is up to the application developer to assign the appropriate attributes to the object so that it may be accessed properly.

This approach assumes that the structure of objects stored in the database is known to the compiler; in fact, the object storage mechanism is compiled and linked with the application program. It also assumes that the persistent objects have a standard set of methods for storing, retrieving, inserting, ordering, etc. These methods constitute the interface to the database and must be chosen carefully to allow replacement of the database structure at a later date.

One function of the database is to provide a simple way to keep track of the images, features and other objects in the system. Each object is assigned a set of identification information, source, dates and data characteristics. This information is stored along with the corresponding object. The database allows users to access and keep track of all images and to select subsets of these images according to various selection criteria for viewing or processing.

The index or ordering information on objects in the database is provided by special objects that have the appropriate structures. Any group of objects may be ordered using this object type by creating an appropriate indexing object. These indexing objects include: binary trees, quad trees, oct trees, hash tables, etc. The indexing objects access the ordered objects by providing an offset into a table storing the data for these objects.

Another function of the data base is to store information derived from the objects. These are arbitrary sized objects and may be retrieved by a variety of attributes. The attributes of these objects are defined by methods on the objects. These methods/attributes may be precomputed and stored in some kind of indexing list or they may be computed when the query is made.

5.4 Processing History

Another database function is to store the history of processing performed on the various image objects. This capability is similar to the source code control system used by software developers. The history system maintains a complete version of the image object along with enough information to reconstruct any other versions of the image object. The processing performed to extract any information (features, etc.) is recorded so that it may be reconstructed if desired. This capability allows the user to use the data from a previous step and to try alternative processing sequences to arrive at new conclusions. All such processing paths are recorded.

The history management system records the sequence of operations that are performed on each image. The original image (or image object object) is stored and the commands with relevant parameters are recorded for each sequence of operations. This is sufficient to recreate the results of any processing sequence. In addition, the intermediate results may be stored, if the user elects to do this.

6.0 Code Libraries

Libraries that support model-based reasoning and signal/image interpretation are built to manipulate the core objects described in section 3. The interface to each function is composed of core objects as far as possible, and the simplest and most general core object as possible to maximize reusability. For example, a two-dimensional window-based transform is designed to operate on a stream of linear arrays. Then any regular two-dimensional container, whether it is embedded in 2-, 3-, or N-space, can be accessed as a stream of linear arrays, and piped into the transform. The core objects support any reasonable data conversions, with priority given to conversions that can be done by a "forgetting" mechanism (e.g. forgetting a linear collection of integers was constructed as type 3d array).

Libraries are themselves a hierarchy of sub-libraries, allowing the incremental inclusion of software packages into applications. The set of possible functions have been grouped into five top-level categories:

Sensor and Image Processing Library
Model Library
Matching and Grouping Library
Interpretation Library
Reasoning Control.

Each library at this level corresponds to a processing stage and/or subsystem in an IU or decision support system. The sensor and image processing library is a collection of objects and routines that process data received from an external source, emulating the interaction of sensors with the objects defined in the input data and/or applying various filters and feature extraction. The model library consists of objects and routines for modeling the world spatially, temporally, and in any other relevant qualitative or quantitative domain. The matching library comprises routines for matching a model-based prediction of the world to sensor-derived evidence of the world.

The interpretation library is a collection of approaches for classifying objects that have been isolated from sensor input and/or mechanisms for inferring decisions. The reasoning control library is a collection of techniques for controlling the execution of this classification/inference process.

6.1 Sensor and Image Processing Library

These are image and signal processing routines and sensor modeling objects. They can be categorized as follows:

Sensor Modeling Objects
Pre-processing (spatial transformations)
Segmentation (extraction of higher-level structures)
Feature Calculation

Sensor modeling objects emulate sensors in predicting the interaction of energy with objects representing matter and energy in the world. The pre-processing functions are operations on 1, 2, and 3-dimensional collections of sensor values that transform, but typically do not reduce the information content of the data (e.g. look-up-tables). Segmentation involves extracting higher-level structure from this data, such as extracting 2d curves and surfaces from a 2d image. Feature calculation computes statistics on these higher-level structures, resulting in collections of measures.

Appendix B lists the public-domain source packages we have identified for further consideration for inclusion in the environment. We also list source packages that are available for one-time license fee with no royalties on reuse of object code.

6.2 Model Library

This library supports the representation and manipulation of spatial, temporal, and other models. Spatial models are typically 2 or 3-dimensional objects, represented as a discrete collection of pixels (or voxels) or represented symbolically with an equation and/or algorithm that describes a shape. Connected models are built out of simple models, linked with constrained coordinate transformations (through methods of coordinate objects) that define the degrees of freedom between the two parts.

The library supports the projection of these models onto other coordinate systems and lower dimensions. 3d models are projected onto a 2d screen. The library also supports the symbolic manipulation of symbolically-represented models, in order to project the model's 3d constraints into a predicted 2d view.

6.3 Matching and Grouping Library

This library consists of routines for placing higher-level interpretations on information extracted from sensor data, by exploiting information stored in models, and matching what is known about the models to what has been observed in the sensor data.

Predictions of what might be seen in the sensor-data constrain interpretation of the sensor data, and, conversely, evidence accrued from the sensor data, directs the consideration of possibilities in the model. Perceptual grouping is included in this library, because it relies on a priori facts about world structure in order to decide how to perceive sensor data.

6.4 Interpretation Library

This library supports various approaches for machine inference. Typically the inference to be made is the classification of some object that has been pre-processed by the sensor and image processing, modeling, and matching/grouping libraries. This library supports multiple inferencing approaches, including statistical classifiers, bayesian inference, logic engines and neural nets.

6.5 Reasoning Control

This library is the control layer for the inferencing that is represented by the interpretation library. It decides what (abstract) processing task to do next, using a particular strategy such as weighing the expected cost, the expected value of information, etc.. and using this metric to choose the next best task to attempt.

Note that reasoning control is not the same thing as process control described below. Process control is a mechanism for sequencing processes using standard programming techniques, and has no special knowledge of decision making techniques.

7.0 Implementation Issues

This section describes the software development and test plan including the management procedures in Section 7.1. Section 7.2 discusses some of the implementation specific issues, that is, issues that are specific to a particular choice of hardware or support software package.

The order of implementation is to produce some core objects and their display methods first. Polygon objects (1,2, and 3D) are the first core objects implemented. Then, the following may be performed in parallel with appropriate interactions to provide a consistent system.

1- User Interface
2- Function/Object Libraries
3- Inferencing
4- Database System

The run-time environment grow sout of the user interface and is developed after the remaining functional areas are essentially in place. This is a recursive/parallel approach in that the four functional areas can be done essentially in parallel with each area being recursively developed to provide additional capabilities. Several design issues that impact the efficiency of both the design/build process and the resultant environment are addressed here.

7.1 Software Management Procedures and Tools

7.1.1 Development Code and Document Version Control

The code and documents for this project are managed using the RCS code control system. This means that all versions are recorded and can be recovered. It also means that the object codes that are created can be strictly controlled. This is important for achieving reusability of code.

7.1.2 Library Maintenance and Installation

The key issues are standards for adding code to libraries, and methods of adding new (sub-) libraries. Standards are addressed primarily by documentation and version control.

The basic method of adding a (sub)library is to extend the class hierarchy to create objects that appropriately utilize new packages of methods. So, for example, multiresolution pyramids are a subclass of connected objects. multiresolution search methods can be added to the objects, or stored in a new sub-library of the matching and grouping library that deals with multiresolution structures.

To aid installation of new libraries, graphical, table-based installation of graphical programming options (e.g. new object creation menu) are provided. In addition, how-to source-code examples, an installation manual, and appropriate make files are provided.

7.1.3 Testing Approach

A public-domain test interpreter, icp, can be applied to each C++ module to test its functionality. Manual code walkthrough after development and initial testing looks for exceptions such as poor memory allocation, inappropriate function calls, missing methods, and unexpected interactions between objects. Integrity testing will be built in to a limited extent, such as code within an object that checks internal pointers for consistency, or writing a known pattern in "dead space" at the end of data structures to catch overwriting.

7.2 Windows Interface

The Xwindows interface is through the InterViews package. This package provides a C++ toolkit for using Xwindows.

7.3 External Data Base Interfaces

There are three aspects to an external database interface: formulating a query that is understood by the serving, external database, communicating with the database to input the query and receive the returned data, and third, handling the returned data.

Most external databases of interest at this time are relational databases that speak standard query language (SQL), such as Sybase, Oracle and Ingress. A graphical interface is formulated that represents data objects as icons or text, and uses graphical selection methods to indicate ANDs, ORs and NOTs, such as pluses between icons for ANDs, an oval enclosing multiple icons for ORs, and the classic diagonal red line for NOT. The graphical query is translated to SQL. This approach gives a generic SQL interface.

The interprocess communication between the serving database and the client IU environment is managed by internet and Unix protocols, with the IU environment running the, possibly networked, database query in the background.

Large volume data returns are handled by two methods: a trap at the internet protocol level to threshold the amount of data that the system is willing to receive (this level can be user set), and a paging/buffering scheme to handle large images and large numbers of images.

7.4 Efficiency Approaches

Operations within the vision environment can be accelerated by three methods: addition of various hardware accelerators, multiple processes communicating via shared memory and other Unix inter-process communication mechanisms, and the networking of multiple platforms. Each of these are discussed in the following subsections.

7.4.1 Hardware Accelerators

The obvious accelerators are relatively inexpensive boards that are Sun compatible, such as the one provided by Vitek. These boards are treated as internal compute-servers, and interaction with them is via a prescribed set of function calls. The function calls accomplish both the download/upload of data, and the setup and invocation of computation.

7.4.2 Shared Memory

Multiple processes can be used to accelerate operations by allowing a compute-intensive operation to be spawned as a background process (or be put in the background automagically when the user begins interacting), leaving the foreground processing available for user interaction. In addition, the time spent waiting for disk I/O and other hardware interactions can be reapplied to other processes.

7.4.3 Networked Platforms

An alternate to add-in boards is a network of machines, some of which act as compute-servers. These machines are dealt with in much the same manner as the add-in boards (see section 7.4.1), with the added possibility of file sharing through use of the Network File System (NFS) standard. In this case, the network is being used to farm out specific computational chunks, rather than being a full distributed problem decomposition. One possible implementation is by remote procedure calls (RPC)s as background processes.

8.0 Applications

8.1 System Engineering Application Development Approach

A high-level approach to generating requirements for the core and domain specific workstations (ws) is shown in Figure 8. Each domain area defines a set of tasks that correspond to client solutions to representative domain specific problems. Requirements are generated specific to each task that would result in a solution if built to spec. These are sifted together to take our best cut at the common core requirements. The generic, core ws is designed to meet these requirements. Based on the core design, extensions are designed to fulfill the domain specific requirements for each domain task. Of course, domain ws should strive to maximize synergy between solutions to multiple problems in the same domain (instead of developing many diverse ws to fulfill diverse tasks). Some of these steps, especially the last three, can be performed in parallel, but the sequential flow picture captures the philosophy of the approach.

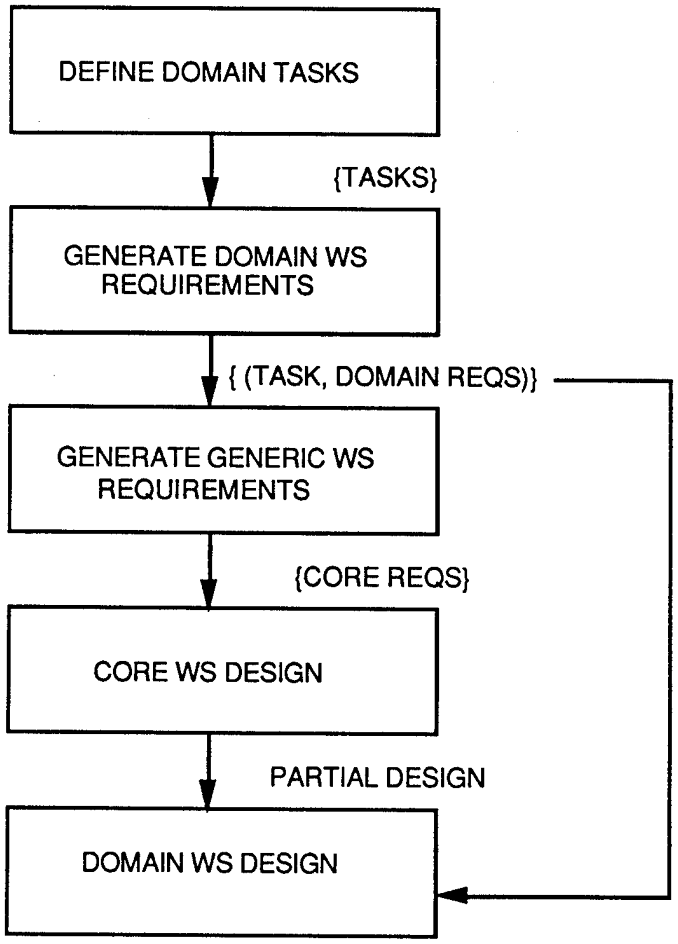


Figure 8: Approach to Requirements

The objectives of an IU Workstation are to provide a hardware and software environment that greatly facilitates development of IU application systems by providing a core infrastructure of integrated tools that are common to most applications, and that traditionally take up the lion's share of development time when systems are built from scratch.

We believe the most direct path to defining such an IU environment is to define domain specific IU task applications and design the IU workstation environment so that it supports development of systems that can perform the domain tasks. In other words, the IU environment is used to build systems that then solve domain problems, but we begin with the domain problems so that we are sure the resulting environment can solve those problems. In order to guarantee building an IU workstation environment with suffcient generality to apply to IU development in many domains, we have adopted the approach pictured in figure 8 to generating requirements for the IU workstation.

Diverse domain areas are chosen and representative tasks are defined in each domain area. Requirements are generated specific to each task that would result in a solution if built to spec. These are sifted together to take the best cut at the common IU workstation requirements. The generic, core IU workstation is designed to meet these requirements. Based on the core design, extensions are designed to fulfill the domain specific requirements for each domain task. We have selected two domain areas, terrain analysis and medical imagery analysis, to focus our workstation development.

8.2 Domain Task Analyses

As explained in section 8.1, two task areas, terrain and medical IU applications, were selected as foci to guide workstation development. The terrain task analysis is presented in section 8.2.1 Section 8.2.2 presents the medical imagery task analysis.

8.2.1 Terrain Task Analysis

We have selected two terrain domain tasks: semi-automated, syntactic, "snap-to" digitization of hardcopy maps and interactive, semantic map/image measurement tools. The terms used in these task descriptions deserve some explanation. "Semi-automated, syntactic 'snap-to' digitization" is the task of interactively choosing a small set of points on a terrain feature from a bit-mapped image of a scanned (i.e., a "digitized" image in the conventional sense) hardcopy map and having the workstation infer the full set of points on the feature (a road, for example), but without specialized knowledge about the feature, except perhaps for basic geometric properties (e.g. linear feature, area, etc.).

An "interactive, semantic map/image measurement tool kit" is a set of measures that can be used on already digitized maps (i.e., scanned and interpreted maps, stored in a digital terrain database), but can also be used at the time of digital terrain database creation to compute and store
attributes. Potential tools include line of sight calculations, earth/hole volumes, shortest-path computation, time-to-travel-path, 3D view visualization, etc.

Roads are a good first focus for semi-automated map extraction, both because they are one of the simplest map features to define for machine segmentation, and because they are required for many common applications. Similarly, line-of-sight analysis is a good first choice for measurements, because it is commonly used, and because it requires manipulation of elevation maps, which is technically difficult to do without computer aid.

Examples of features stored in digital terrain databases and associated textual (usually relational) databases that are candidates for semi-automated extraction include roads, railroads, hydrology features, land parcels, agricultural plots, fields, forests, task-defined elevation features (e.g. hills, air-space obstructions), and task-defined sets of buildings (e.g. residences, industries). Examples of measurements made from digital terrain databases include, line-of-sight, land volume, areas of different (and possible compound) terrain types, watershed, distances between points and between geometrically more complex terrain features, such as rivers and towns, as well as topological relations for different terrain features including between, nearness and adjacency.

The objective of terrain task analysis is to decompose the execution of the terrain tasks such that the underlying required functionality is naturally exposed, making more obvious the design solutions to building a system that achieves this functionality. Our approach to this is twofold: first to step through the system user's tasks in scripts, essentially scripting (or verbally storyboarding) the functionality of the user-interface, and second to analyze data transformations as the user experiences them (based on the storyboard) inferring progressively finer levels of black-box functionality the system must possess. The terrain task script is presented in section 8.2.1.1, and the user-apparent data transformations are discussed in section 8.2.1.2.

8.2.1.1 Terrain Task Script

1) Select map, scan (i.e. bitmap digitize) if necessary.
Assumes: Maps exist in an on-line accessible form, and/or there is a user-friendly scanning procedure available for hard-copy maps. The latter also assumes that scanning resolution is sufficient versus map detail to enable the feature selection, etc. of the following steps.

2) Indicate feature type (e.g., linear, area, road, field...) or measure type (e.g. line-or-sight, shovel, hourglass, spyglass...).
Assumes: Feature and/or measure types are relevant to map features and to the domain task (e.g. urban planning, agricultural survey, etc.)

3) Select feature or measure type.

4a) Select mode to label feature and type-input for feature label.

5a) Choose points on features in displayed map.

6a) Select mode to do snap-to feature segmentation or to perform the measure.

7a) View the displayed results of feature extraction and/or measurement.

8a) Modify the displayed results as necessary.

9a) Select mode to store labeled features or measures to database.

Assumes: Storage mode is a known default. Otherwise this step must be enhanced to select storage location. Typically, the map and/or measure type will have a storage location and format associated to it already.

4b) Select choose feature mode and type-input for label for the feature.

5b) Choose points on features in displayed map.

6b) Repeat 4b and 5b until all labeled features selected for extraction or measurement.

7b) Select batch-mode for snap-to feature segmentation or measurement.

8b) Select mode for viewing results of batch feature extraction or measurement.
Assumes: Job is identified by user or other method and accessible by that identification. If user has multiple jobs, system must appropriately differentiate and allow user selection for viewing.

9b) Modify the displayed results as necessary.
Assumes: All results of this job are simultaneously displayable and usefully viewable (e.g. non-overlapping). If not, steps 9-10 must step through feature by feature or screen by screen.

10b) Select mode to store labeled features or measures to database.
Assumes: Storage mode is a known default. Otherwise this step must be enhanced to select storage location. Typically, the map and/or measure type will have a storage location and format associated to it already.

8.2.1.2 Terrain Task Data Transformations

A terrain task data transformation refers to the displayed output a user observes in response to a set of inputs while performing the terrain task script. Inputs include items selected or typed by the user, as well as data the user assumes is present, such as scanned maps and digital terrain databases. The "transformed data" includes the visually observed displays, such as a bit-map overlay of an extracted terrain feature, as well as the implied data the user assumes supports the display, such as the bounding polygon of the segmentation. The user-apparent data transformations suggested by the task script include the following.

1) selected and typed entries -----> map (= scanned map image + any associated textual data)
2) selected map -----> displayed map and textual data
3) selected feature type and chosen points -----> segmented feature
4) feature segmentation -----> feature segmentation (interactive)
5) selected feature type and chosen points -----> segmented feature + measure
6) selected extracted features and/or measures + labels-----> database record (indicating storage of features and/or measures)

The transformations imply the existence of certain user-apparent system functionalities. These are listed below.

<u>Transformations (1) and (2)</u>

A database(-like) facility is implied that consists of a set of storage units for scanned maps imagery and associated textual data, with the database facility inverted on one or more of

a) map type
b) measurement task type
c) map source
d) digital database for map feature and/or measurement storage
e) job identifier

For single-task users, like urban planners, the measurement task type or map type are more likely keys. For multi-task users, like DIA, or a centralized city GIS facility, the map source and/or storage database are more likely keys. In the former case, the user usually accesses the same set of maps, extracting information and making measurements from it. In the latter case, the user accesses from a wide variety of maps and map databases, often updating the maps and associated databases. Job identifiers can be used to allow individual users or teams to work incrementally on an on-going task. A job has an associated data structure that saves pointers to the input and output data sources and job state. They can be indexed by user names or task names.

Transformation (3)

This is the critical technical step of semi-automated segmentation. The feature extraction itself implies a specific choice of segmentation methodology, and the use of specific image processing, pattern recognition, search, and inference procedures. The specific choices are not user-apparent. However, the output display is user apparent and must allow the user to "verify at a glance" the correctness of the terrain feature extraction. This probably requires a side-by-side display of the system segmentation in graphic overlay, next to the un-segmented scanned map.

The choosing of some points on the terrain feature in the displayed map serves the two purposes of indicating which feature is being stored and associated to the typed-in label, and for seeding the otherwise automated feature extraction process. For example, to semi-automate road segmentation in a scanned map, the user could be directed to choose the two endpoints of the road segment, then the system extracts the road segment between the two points. The choice of points clearly depends on the feature extraction task; the user must be directed how to choose points so that they are meaningful to the system with respect to the user's task. On-line instruction regarding geo-object choices for the various feature types and data sources should be available.

Transformation (4)

The user views the system's display of its terrain feature segmentation results and allows the user to interactively edit the results if necessary. This is done using the operations add, delete, move, undo and save, as defined in the glossary. The regions and/or geo-objects should snap-to as points or other geo-objects are added or deleted. A reasonable first definition of snap-to is gotten by using shortest-planar or shortest-elevation-grid distance to link points or other geo-objects.

Transformation (5)

A measure requires dynamic extraction of terrain features when they are not pre-stored. In either case the system must understand a priori what features are required. The points chosen by the user define the geographic region of interest for the measure. If feature extraction is necessary, the user should be cued appropriately. The system should be set up so that it is easy for the user to abort the process if it requires feature extraction s/he does not wish to do.

Measures can output complex data structures such as geo-objects and/or regions, as well as textual and numeric outputs. For example, line-of-sight calculations output a polygon or elevation region with line-or-sight from a point or other region. Land volume, on the other hand, outputs only a single number. Some measures require sohpisticated geometric computations, such as surface, polygon, and/or line intersection, union and difference. A full polygon algebra should probably be supported.

Transformation (6)

Status messages for intialization and completion of tasks should be displayed. Storage and retrieval of large volumes of data should have displayed meters or other devices to indicate that the system is engaged in a task and how close to completion it is. Storage and retrieval times should be estimated and warnings issued for time consuming processes allowing the user option to alter or cancel the storage or retrieval command. Asynchronous, batch-mode storage and retrieval should be supported.

Systems that provide multiple users to cooperate on the same task may have special requirements for data integrity, such as write-lockouts to avoid synchronous map updating.

8.2.1.3 Terrain Applications Requirements

In both terrain tasks, the top level user interaction steps are as follows:

1) Select maps/imagery, scan (i.e. bitmap digitize) if necessary.
2) Indicate feature type (e.g., linear, area, road, field...) or measure type (e.g. line-or-sight, shovel, hourglass, spyglass...).
3) Select points on features in map and/or image.
4) Tell system to input features to database or to perform the measure.
5) Verify correctness of feature extraction and/or measurement.

Infrastructure requirements such as friendly and efficient user interfaces, a need to robustly interact with digital terrain databases, large data storage and retrieval capabilities, etc. are common to more than one step. Nonetheless, in the following, we attempt to list requirements in correspondence to the user functions, rather than in terms of elements of the solution (like user interfaces).

1) Select maps/imagery, scan (i.e. bitmap digitize) if necessary.

1.1) The system must be equipped with standard map scanning (e.g. flatbed digitizer) interfaces that provide adequate resolution to capture the necessary map detail.

1.2) If scanner is used, then digitizing should be done on screen on the scanned map.

1.3) Sufficient workstation memory is required to hold the terrain database corresponding to the scanned map segment in memory.

2) Indicate feature type.

2.1) The system must interface to ARC/INFO, and a selection of standard terrain digital databases to be determined by the target market.

2.2) We should choose and make available a digital terrain database that comes standard with the product. Coverage and resolution requirements will be determined by market analysis.

2.3) A uniform interface should be available for all terrain databases the system communicates with.

2.4) A display showing the selectable terrain features present in the database corresponding to the displayed map segment should display in less than 10 seconds from the time map coordinates are entered, if the database segment is in local storage.

2.5) If the digital terrain database segment requires more than 15 seconds for retrieval to display, a message should be displayed telling the user the function that is being performed, and giving dynamic indications of progress.

2.6) The user should be able to select the digital terrain features with a minimum of manipulation (e.g. keystrokes, mouse clicks, etc.).

3) Select points on features in map and/or image.

3.1) It should be clear how to enter the modes to indicate points.

3.2) The points should be coded so that it is clear which correspond, to which features and/or measures; it should be intuitive and require minimal manipulation to change points, "move" a point in either map or image, etc.

3.3) It should be obvious how to indicate that point selection is complete, either on a feature, or on the entire image/map.

4) Tell system to input features to database or to perform measures.

4.1) It should be obvious how to indicate that point selection is complete.

4.2) The system should return control to the user in less than 10 seconds; either input and/or measurement should be available for verification or relegated to an off-line or background process.

4.3) If producing the verification display needs to be a background process, then there must be a protocol of queuing and recalling verification displays at a later time (e.g., stored up in a pull-down list). Any job that has been completed as far as the user is concerned should appear in this list, with its status indicated even if it's not yet available for verification.

4.4) If any process that occupies the user interface (so that the user cannot get response from the workstation) takes more than 10 seconds, a message should be displayed indicating operations in process.

5) Verify correctness of feature extraction and/or registration.

5.1) A verification display should be available showing an overlay of extracted features against map and/or imagery features. This should be togglable so that the user can switch between the original and extracted in focus of attention.

5.2) It should be possible to interactively correct the extracted feature; re-storage must adhere to the requirements in (4).

5.3) Correction of feature extraction should be intuitive; it shouldn't be necessary to read a manual.

8.2.2 Medical Task Analysis

We have selected two tasks: automated intercortical volumes from hand radiographs and automated prostate volume quantification in CT imagery. Both measures are key medical indicators; the first in diagnosing and tracking arthritis, the second is a direct indicator for prostate surgery (radical prostatectomy).

The choice of the arthritis measure application is motivated by the following facts. The physics of sensor interaction of radiographs with human hands is a completely modeled, well-understood technology. Probability models for scattering of xrays in soft-tissue, bone, and air are commonly available in the medical physics literature [Curry et. al.-84], [Kereiakes et. al. -86]. This forms the basis for strong prior probabilities in imaging models. In the application of radiographs, the imaging geometry, approximate object (i. e., hand) aspect, and the ambient characteristics of the energy source (i.e., xray voltage) is always known, so this is a highly constrained, but not a toy problem. In this respect it is representative of a broad class of medical, manufacturing and inspection tasks for machine vision. Finally, hands are complex enough that the modeling problem is important, but simple enough that we can hope to accomplish the task within a reasonable project scope. Hands are 3D with articulated joints. Because the sensor is invasive, it is necessary to model the layered volume (not just the surface). Most of the primitive hand components are cylindrical in basic shape [Meschan-75]. We anticipate that it is not necessary to model deformable surfaces for hand recognition. Some population statistics for normal variation in bone size and range of joint articulation are available in the medical literature [Poznanski-74].

Prostate volume measurement requires working with 3D CT imagery, requiring image processing operations to exist for 3D image processing, and similarly requiring surface matching and volumetric understanding as is required in range imagery. The prostate requires irregular curved surface matching, but having genus one, is much simpler than the heart, for example. Additionally, the prostate is a static organ, unlike the heart, so that shape remains stable over time.

As in the terrain task, the objective of medical task analysis is to decompose the execution of the medical tasks such that the underlying required functionality is naturally exposed, making more obvious the design solutions to building a system that achieves this functionality. We again present a task script, section 2.2.1, followed by the medical task data transformations in section 2.2.2.

8.2.2.1 Medical Task Script

1) Order exams/measures

1.1) Select order mode

1.2) Select appropriate fields (e.g. modalities, views, measures)

1.3) Type input for name, institution id OR use OCR to scan patient demographic data

1.4) Select results viewing station(s)
1.5) Select execute mode

2) Call up results
2.1) Select results mode
2.2a) Type input or select "my" icon to call up batched exams
Assumes: batched exams and "my icon" exists
2.2b) Select appropriate fields
2.3b) Type input for name, institution id OR use OCR to scan patient demographic data

3) View results
3.1) Select exam(s) to view
3.2) Select retreive and/or display mode

4) Modify results
4.1) Select modification mode
4.2) Select sub-mode of delete OR add OR move OR undo
Assumes: modification mode selected was "segmentation"
4.3) Choose geo-objects
4.4) Select done-modify

5) Order additional measures
5.1) Select measures mode
5.2) Select appropriate fields
5.3) Type input as required by 5.2 (typically none required)
5.4) Select results viewing station(s) (should default to current settings)
5.5) Select execute mode

6) Archive results
6.1) Choose exams to archive
6.2) Select archive mode
6.3) Choose archival devices
Assumes: More than one archive available
6.4) Select execute mode

8.2.2.2 Medical Task Data Transformations

A medical task data transformation refers to the displayed output a user observes in response to a set of inputs while performing the steps in the medical task script. Inputs include items selected or typed by the user, as well as data the user assumes is present, such as medical imagery from exams, and demographic data routinely associated with the exams. The "transformed data" includes the visually observed displays, such as a bit-map overlay of a segmentation, as well as the implied data the user assumes supports the display, such as the bounding polygon of the segmentation. The user-apparent data transformations suggested by the script include the following.

1) selected and typed entries -----> exams (= imagery + textual data)
2) selected exams -----> displayed imagery and textual data
3) selected and/or displayed exams -----> segmented imagery
4) segmentation -----> segmentation (interactive)
5) selected exam -----> measure
6) two selected exams -----> comparison measure

The transformations imply the existence of certain user-apparent system functionalities. These are listed below.

Transformations (1) and (2)

A database(-like) facility is implied that consists of a set of storage units for exam imagery and textual data, with the database facility inverted on

a) patient names by lexicographical ordering
b) patient by social security number
c) patient by local institution id
d) anatomy by hierarchical anatomical structuring order
e) dates by chronological ordering
f) modalities by lexicographical ordering OR data storage source
g) non-local institutions by lexicographical ordering.

The options are in priority order for the medical task. Most access is concerned with specific individuals, however, doctors also need to be able to access on anatomy and modality for purposes of teaching, research and demonstration. The options (e) and (f) are probably not both necessary to invert on (because the bucket size of retrieved exams will be small after patient and/or anatomy and/or either one of modality or date are selected), but are included for completeness. The option (g) will only be applicable when multi-site institutions are involved, such as the VA, Humana and Kaiser medical centers, hospital networks in socialistic countries (as most are run in Europe, for example), DoD hospitals, distributed clinics, etc.

Because of the large size of imagery in exams (typically from 6 to 100 megabytes per exam), it is standard practice to separate the textual data from the imagery, and to make it available from database queries without requiring associated imagery retrieval. The non-imagery exam data includes all textual and numeric patient data, including demographic data, dates and locations of visits, attending and referring physicians, modalties of imagery, anatomy imaged, imagery numbers, sizes, bit-depths, and pointers, diagnoses, measures, and additional physician's comments.

Transformation (3)

The capability to segment imagery implies a specific choice of segmentation methodology, and the use of specific image processing, pattern recognition, search, and inference procedures. The specific choices are not user-apparent. However, the output display is user apparent and must allow the doctor to "verify at a glance" the correctness of the segmentation. (FYI, doctors often call segmentation "contouring" imaged tissues. Bones, organs, and other soft tissues like ligaments, fat, etc. all fall within the taxonomic category called "tissues".)

Transformation (4)

The user (a physician or highly skilled technician) must be able to work directly on the verify-at-a-glance displayed output of transformation (3) to modify it for any system created segmentation errors. The operations of add, delete, move, undo and save are the tools for this. The user should be able to push the display around until s/he likes what s/he sees, and then save the result, which will be used for auto-recalculation of diagnostic measures that depend upon the associated segmentation.

Transformations (5) and (6)

The basic measures are area and volume of segmented tissues. For an area, this is computed as pixel count multiplied by the appropriate (constant per image) factor that transforms pixels to square centimeters based on the viewing geometry and physics of the scanner. For computed radiography (digital xray) we can automagically measure this factor from the grid that appears on the borders of the image. For digitized imagery, we must input this factor based on specific scanners and viewing procedures, or we must automagically measure it from a marker of known size placed in the image. The standard such marker is an "L" or "R" that is routinely placed on most xray imaging plates to indicate the view the patient was imaged from.

Other associated area measures include longitudinal axis computations for phalanges, average pixel intensity, maximum region diameter, and specific region diameters that depend upon finding certain anatomical points in the bounding polygon of the segmented region.

Volumes are usually computed by doing the areas of the slices, and then interpolating between adjacent segmentations. The interpolation is straightforward, but depends upon knowing the thickness of, and distance between, slices in the exam (as the set of images making up the volume is called). Usually these are constant factors for a given exam, but they do not have to be(!).

Comparison measures usually compare the measures the last time the patient was in against the current measures. So this requires the system to call up the last exam, make the measures if they weren't made then, and then also to do the measures on the current exam. Comparisons will typically be presented as percent increase or decrease, as well as absolute increase or decrease. If there is a longer history, another presentation, such as a graph, might be nice. Physicians do not currently produce such graphical aids. Comparison exams should be presented side by side, with the old exam to the left of the new one.

8.2.2.3 Medical Applications Requirements

The medical workstation design requirements are driven by the workstation task requirements, and divided according to a top level breakdown of functional units including:

- Physician/Technician User Interface
- Network to Scanners, PACS, RIS, and HIS
- Databases: Anatomical Models/Scanner Models/Patient Exams/Imagery Features
- Image Processing Functions
- Inference for Anatomical Segmentation
- System Control.

In the following, we break out requirements according to this functional system decomposition.

1.0) Physician/Technician User Interface

1.1) The interface must be highly reliable and fault-tolerant. In particular, it should be possible to move between states without rolling back through each intermediate state, and to alter inputs and/or abort at almost any point, saving state(s) and without catastrophic consequences.

1.2) Manipulation (typing, pointing, etc.) should be minimal to accomplish any task.

1.3) There should ideally be only one visual focus of attention at any time on the screen.

1.4) It must be possible to paint an image in less than 5 wall-clock seconds. In general, no display should be slower than this (graphs, spreadsheets, etc.)

1.5) Displays must support a minimum of 16 bit deep pixels.

1.6) Full color overlays must be provided. Color should be 24 bits deep (i.e. 8 bits each of red, green and blue) and color-tables should be changable in software.

1.7) It should be possible to draw on an image, then erase the overlay without (apparently) affecting the image underneath. In particular, if the image needs to be re-drawn, it must be transparent to the user.

1.8) Both interactive and automatic imagery color overlay capability should be provided. Color tables should be interactively changable.

1.9) Zooming should be accomplished in no more than 5 wall-clock seconds. Both zoom window and zoom around point should be supported. Zooming should zoom both image, overlays and any "chained" windows.

1.10) Both 2D and 3D imagery display should be provided. 3D imagery must support 512x512x256, 2D must support 4Kx4K imagery. Imagery must be viewable in its entirety (i.e., without scrolling) in unzoomed mode.

1.11) Volumetric imagery must be viewable from any perspective. 3D and 2D rotation and translation must be supported.

1.12) 2D cutaway views of 3D imagery at an arbitrary angle should be supported.

1.13) 2D imagery display should "fill" the display window.

1.14) 3D graphic object display should be supported including raster and vector representations with hidden line and surface capability.

1.15) It must be possible to overlay 3D graphics on 3D imagery, with the same erasability requirements (see 1.7 and 1.8) as 2D overlays.

1.16) Display should convey the same information to colorblind people as it does to normal color-sighted people.

1.17) Any operation requiring more than 5 seconds of wall-clock time to execute should display messages indicated what it's doing.

1.18) Any operation requiring more than 15 seconds of wall-clock time should be provided in background mode so that other workstation interaction can go on.

1.19) Imagery displays should support mouse, light-pen or other pointer interaction.

1.20) Menus, sliders, buttons, tables, graphs, icons and sprites should be supported both interactively and automatically.

1.21) Imagery browsing should be supported with programmable choice of "sub-sample" function for reducing imagery size for browse windows.

1.22) If multiple screens are required to browse through a single patient exam (which can be up to 256 images), screen switching should be extremely rapid, certainly under 5 seconds, hopefully under 3 seconds.

1.23) The user interface should be easy-to-learn, easy-to-recall and easy-to-use.

1.24) Custom interface selections (e.g. customized defaults, button locations, etc.) should be available and easy to set up and use.

1.25) An easy to use access-security system should be provided. This security system must allow programmability of selection of access to system functions/modes/devices/data that are permitted for varying levels of clearance.

1.26) The user interface to external databases, etc. should be represented in visual, medical-domain terms/appearance.

1.27) The user interface should be portable to (almost) any Unix box with sufficient memory. It should provide network server capability (a la X, NeWS, etc.)

1.28) On-line help should be available but not needed by the user.

2.0) Network to Scanners, PACS, RIS, and HIS

(PACS= Picture Archival and Communication System, RIS= Radiology Information System, HIS= Hospital Information System)

2.1) The workstation must support standard networks, (ethernet, fiber-optic ethernet, hipi, etc.) and standard network communication protocols including TCP/IP, ISO, IEEE, etc. Custom networks must be supported for critical vendors (probably PACS systems not known at this time, scanner vendors we contract with, etc.)

2.2) High-speed/high-bandwidth communication must be supported. Exact numbers are not yet known, but probably in the hundreds of megabytes per second range.

2.3) Must decode and read imagery formats including ACR/NEMA, ISO and IEEE, as well as "proprietary" formats for CR/CT/MRI scanners made by GE, Siemens, Philips, Toshiba, Picker, and Diasonics at a minimum. Data structures, selections etc. should support addition of formats as required.

2.4) Data exchange between internal databases and institution databases must be supported including all leading commercial PACS, RIS and HIS systems.

2.5) We must provide accurate modification tracking of changes to medical records, including noting who made what changes when.

2.6) The possibility of distributed updating of records must be accounted for, either by some communication protocol between workstations, or by prohibiting it (e.g. data locking).

2.7) Access to records must be security controlled to meet legal, medical and institutional requirements for patient privacy and medical protocols.

2.8) The workstation should be able to obtain dynamic models of the contents of the accessed external databases such that if an unusually large retrieval is indicated, the retreival is confirmed before execution.

2.9) The workstation should provide a uniform interface to all imagery/records storage devices. Details should only be available for debugging problems; otherwise it should look like a single database.

2.10) If multiple workstations are networked in and/or between institution(s), they should communicate to effect transfer of data to meet scheduling needs for availability of records/imagery where and when required for operational needs of the institution(s).

3.0) Databases: Anatomical Models/Scanner Models/Patient Exams/Imagery Features

We first list requirements that are common between databases, then the requirements that are specific to each.

3.1) We need to be able to query over arbitrary "keys" or access slots.

3.2) We need to be able to construct arbitary boolean algebraic queries over multiple database keys.

3.3) The interface to database queries, whether they are databases resident in the workstation or external to it, should be intuitive and easy to use by anyone with a high-school education (e.g., you shouldn't have to know boolean algebra).

3.4) Database searches should be optimizable (i.e., programmable), to depend on the type and space/time characteristics of the data being searched for (e.g. graph search over anatomic models, spatially oriented search for perceptual grouping in imagery, etc.)

3.5) The anatomical models and scanner models databases must be persistent.

3.6) The patient exams database must be locally persistent until it can be verified that the records/imagery have been archived in the appropriate external database.

3.1) Anatomical Models Database

3.1.1) The database should support representation, storage and retreival of point, curve, surface and volumetric objects, including graphs and boolean combinations of them.

3.1.2) Complex relations such as articulation should be supported between various modeled anatomical parts.

3.1.3) Pointers to image processing and feature extraction operators should be supported.

3.1.4) Models should be indexed by patient demographics. In particular, ranges and probability distributions over them should be representable for all model features.

3.2) Scanner Models Database

3.2.1) Models should be available for computed radiography, digitized xray, computed tomography and magnetic resonance imagery. Scanner modeling includes procedures for predicting appearance of imaged anatomy based on imaging geometry and anatomical parts (as represented in the anatomical models database).

3.2.2) Output of applying a scanner model to an anatomical and imaging model should be an imagery appearance prediction that is represented to be usable by image processing and inference operators.

3.2.3) Distributions of prior probabilities of imagery appearance based on 3.2.2 should be provided in an inference readable format.

3.3) Patient Exam Database

3.3.1) Records selection must operate from all minimally sufficient queries.

3.3.2) Self-contradictory queries must be handled, either by making them impossible, by offering auto-ORing and confirmation to the user, or some other method(s).

3.3.3) Matching of input data such as patient names, social security or other id numbers, etc. should support partial matches, be case insensitive and accept all possible standard syntaxes (e.g. first name, M.I., last name versus
last name, first name, M.I., etc.)

3.3.4) Ideally, mispellings, homonyms, and any other known standard data entry problems should be handled subject to reasonable speed requirements (TBD).

3.3.5) It should be clear if a measure has already been performed, and it should be easy to indicate re-doing the measure.

3.3.6) Multiple values for the same measure should be maintained, and the differences accounted for (e.g., if a physician interactively modifies a segmentation and asks for a recalculation, but all on the same exam.)

3.3.7) It should be easy to retrieve and view prior exams, and indicate which ones should be used for comparison measures and/or trend tracking; again standard defaults should be available and easy to indicate.

3.3.8) Multiple exam trends should be plotted in a self-explanatory, easy to read fashion, with links to multiple records available for display of interactively selected comparisons.

3.3.9) Linked records, e.g. comparison measures across multiple exams, should be indicated so that they can be quickly recalled for viewing.

3.3.10) Sufficient local storage should be provided to buffer exams. Sizes of this buffer are not yet known, but will easily be in the hundreds of megabytes and is likely to be in the thousands of megabytes.

3.4) Imagery Feature Database

3.4.1) Dynamically created feature databases as the output of image processing operators should be supported.

3.4.2) Feature databases should be indexed by the imagery the features were extracted from.

3.4.3) The databases should be spatially hierarchically represented to support optimization of search for feature groupings.

3.4.4) Features should be linkable as instances of anatomical models that are linked, when this makes sense (e.g. spatial relationships between extracted bone surfaces).

3.4.5) Feature databases need only persist until imagery analysis is completed.

4) Image Processing Functions

4.1) A large library of standard image processing functions must be provided, including arbitrary kernel sized convolutions, where the convolution window allows arbitrary arithmetic and logical operations on the neighborhood.

4.2) Arbitrary sub-window and blotch (i.e. mask dependent) processing should be provided for all IP operations.

4.3) It should be selectable on all operators whether the output takes the form of images, lists, or a feature database, where these outputs make sense.

4.4) Operators should be able to accept the output of feature database queries as inputs.

5) Inference for Anatomical Segmentation

5.1) Full Bayesian inference over either continuous or discrete values must be supported.

5.2) Bayes nets must be dynamically instantiatable to correspond to instances of hypotheses of instantiated models.

5.3) It must be possible to query the state of any subset of the Bayes net; reasonable subsets (e.g. subtrees) should be efficiently searched.

5.4) The Bayes net must be a savable structure, but need not, in general, be persistent.

5.5) It should be possible to efficiently search the model space and feature space to perform matching, and to efficiently instantiate Bayes nodes based on the result of the matches.

5.6) Metrics used for matching should be programmable; in particular, sophisticated match metrics such as the Mahalanobus distance should be supported

6) System Control

6.1) A full utility theory over the Bayes nets should be provided.

6.2) It should be easy to select and change value functions over the anatomy of interest. The value functions should be linked to the anatomical model database, and should apply to arbitrary levels of hierarchy there (e.g. curves and surfaces as well as volumes).

6.3) Control should account for listening to the user while still maintaining good efficiency in computationally intense processing.

6.4) The system should understand when "batch" type processing is required and when realtime interaction is required. If it cannot provide the latter, it should inform the user and give appropriate options.

6.5) System control should exhibit a certain level of fault tolerance; in particular, self-diagnostics should be periodically run, and appropriate message displayed if servicing is needed.

8.3 IU Application Development Task Analysis

The objective of IU application development task analysis is to present the generic steps an application developer often repeats, and also to present the dependencies between steps such that the underlying requirements for tool capabilities are naturally exposed, making more obvious the design solutions to building tools and a tool using environment that achieves this capability. Our approach to this is to step through the tasks a "generic" IU application developer performs in scripts of progressively finer detail, essentially scripting (or verbally storyboarding) the functionality of the user-interface. Section 8.3.1 is a summary of tasks the developer embraces in building an image understanding (IU) application. In section 8.3.2 the application developer's script is presented.

8.3.1 Summary of Application Developer Tasks

It is problematic to present a script of the actions an application developer performs, as developers may permute the order and dependencies among operations in many ways as they incrementally interleave development steps such as image processing, model building and matching, for example. So the workstation designer needs to be especially wary of depending too directly on the folowing script as a specification for doing tool development. We can be sure that the functionality listed in this script is a strict subset of what is actually required. Nonetheless, the intention of this script is to cover the main steps and the obvious dependencies between steps in sufficient breadth and depth so that the workstation designer produces a useful environment even if s/he takes a narrow interpretation of the scripted capabilities.

A second complication is the need for the IU application developer to deal (more or less) directly with many of the objects that are created during development sessions. It is tricky to discuss the required functionality without suggested design solutions (e.g. object structures) to simplify the language. However, the attempt is made here to state the functionality without designing solutions to achieve it. As a consequence the script sometimes reads a bit more like a set of tool requirements than a script of actions.

The generic task the script is focused around is that of doing the development to automate an IU application. Basically, the developer (we interchange the terms developer and user from now on) wants to bring up some imagery, interactively play with it to make measurements and begin guessing what sort of imagery operators can work on the data. Then s/he wants to string together a bunch of existing image processing (IP) operators, playing with parameters interactively to see what evidence is extracted from the imagery.

Then begins the task of model development. The models are geometric, material, constraints, and IP (or other) actions for evidence gathering. This requires interactive geometric modeling, integrating population statistics, setting up and experimenting with constraint equations, displaying lots of 2D and 3D geometric objects, and interactively editing to create model objects out of these pieces.

Development of matching operators typically descends into a full programming environment, writing matching routines dealing with the model and IP objects that have been defined. Some basic math packages for solving cubic spline equations are helpful here, and probably similar tools for various other parametrizations. Grouping is a type of hierarchical and/or adaptive combination of searching and matching features into more complex structures. Groupers can be looked at as complicated matching routines, for the sake of how they fit into a development environment. However, they tend to be different in that they can make extensive use of Powervision style image feature filtering over a database of imagery features to do their job. The need to have computationally intense numerical matching operations in a tight loop with database calls that invoke spatial searches for the data to be matched against is unique to computer vision grouping operations (to the best of my knowledge). The idea of implementing this as database filtering is actually a design approach rather than a required capability.

Now the IP operators, models and matching and grouping routines are integrated together in an inference framework of some sort. Three of the most common such frameworks in IU applications are Bayesian inference networks, blackboard-executed frame-based systems, and logical rulebases. In any of these paradigms, the developer defines model representations of (Bayes) nodes/frames/rules that point at the set of model-components that can be confused in recognition. Conditional and a priori probability distributions/confidences or weights of evidence must be defined by either LUT or parametrizations and put into the network model/frames/rules. Models for the IP operations should include expected time of execution as a function of sw/hw environment and relevant input data parameters such as imagery size. The Bayes net/blackboard/inference engine gets cranked manually until inference starts looking reasonable.

When the net seems to operate well with human control, the next step is to experiment with automated control regimes. For Bayes nets, values get assigned to the top level nodes of the Bayes net, and utility functions can be generated from the bayes net. The utility function assigns a number to each action that can be executed as a self-contained process that comes from the bayes net. These numbers can be used to rank the processes for operating system style control like FIFO and best-first. Alternatively, a full, decision theoretic control can be used, or even rules giving an exact specification of steps to be executed in the Bayes net. In blackboard systems and rulebased systems, meta-rules must be developed that guide search routines and prioritize rule execution.

Finally the whole thing's gotta be tested, a lot, by components and by system. Statistics from runs should be automagically accumulated. Timing and other standard software metrics should be provided at a tool level. Tables and graphs should be easy to generate. Then the whole schmeer has to be software engineered for maintenance, documentation and versioning.

8.3.2 Hierarchical Application Developer Task Script

The "script" detailing the above development task summary is presented in levels, each subsequent level expanding the detail of the previous. Assumptions about the mode the system is in and the availability or display of data are indicated for each step in the script where appropriate. Two types of or-ing for interaction options are used. One is versioning indicated by "a", "b", etc. after the step number. Thus "4a" means the "a" version of step 4, and "4b" means the "b" version of step 4. The other type of or-ing is just to put "or" between options within the same script step as in selecting a feature type or measure to perform.

Level 1

1) Access the appropriate data sources from the development environment.
2) Display, manipulate and examine data.
3) Develop image processing operators that extract evidence from imagery.
4) Extract features and measures from imagery regions.
5) Create models of objects to be recognized and measured from imagery.
6) Develop matching operators that compare regions, features and measures with segments of models.
7) Develop inference structures such as Bayesian networks or rule bases.
8) Experiment with reasoning control strategies on the inference structures.
9) Craft the user interface to yield a natural vertical application solution.
10) Test the (semi-) automated solution for robustness and reliability.

Level 2

1) Access the appropriate data sources from the development environment.

1.1a) Type-commands to access images from known locations.

Assumptions: The developer knows the image s/he wants and it is accessible by the system. The system need only have routines to use a pathname to retrieve an image. The system may need to be able to access external databases, such as a digital terrain database, and know foreign imagery formats, such as for medical images.

1.1b) Display an imagery database browser and make selections to retrieve desired imagery.

Assumptions: All display and retrieval interaction is idiot-proofed. For large retrievals, the developer is warned of the amount of data and asked to confirm the retrieval. Interface has full database capability (keys on imagery names, dates, sensor-types, general image content, etc.).

1.1c) Something in-between 1.1a (the user is smart and the system is dumb) and 1.1b (the system is smart and the user is dumb).

2) Display, manipulate and examine data

2.1) Display the selected imagery.

Assumptions: Display function is smart about window sizes versus image sizes. Display does not change aspect ratio of imagery. Clipping is optional.

2.2) Do display manipulations of imagery, at a minimum including scrolling, zooming by pixel replication, anti-aliasing rotation (e.g. the Fant warp routine), fast transpositions and re-scaling.

2.3) Text describing imagery objects or ephemeris data should be able to be moved into windows so it can be side-by-side with imagery or other signals. It should be able to be zoomed up or down or font substituted to be bigger or smaller.

2.4) Imagery should be interactively examinable for gray-level distributions and pixel values in arbitrary sub-regions of the image. In particular we should be able to look at an ascii (numerical) view of sub-regions of pixels, or get the histogram, mean and variance of pixel values in an arbitrary region. It is nice to be able to graph the profile of the pixel values under an interactively defined geo-object-line or for any arbitrary sequence of pixels. It is convenient to be able to drag a small window interactively over the image and see display of pixel values or other statistics dynamically computed and displayed.

Assumptions: These operations can be interleaved with the display and processing of any image. It should be possible to represent sub-regions for display and examination by multiple methods. At a minimum it should be possible to represent sub-regions as either regions output by connected component, or defined interactively as a polygon or other geo-object.

3) Develop image processing operators that extract evidence from imagery.

3.1) Define and move about image display windows easily and have names for them that are usable in interactive operations.

Assumptions: Image operators are smart enough to know about their need for scratchpad memory (or scratchpad windows), and whether operations can be done in place (like LUT filtering) or requires an output window (like FFT). The user is prompted or shutout from illegal operations automagically.

3.2) Do standard look-up-table (LUT) filtering. The developer should be able to easily set a LUT for gray level and/or color either by putting values in a file or an interactive data structure, or by parametrically defining a function to generate values for the LUT. Histogram equalization, parametrized gamma correction, absolute and multi-value thresholding should be available as standard LUT generating routines.

3.3) A generic method should be available to do an algorithmically parallel neighborhood operation at every pixel. The most general capability allows the developer to write any function that reads the values of the pixels in a runtime defined neighborhood (n by m, circular or hexagonal) and replaces with center pixel with a new value returned by the function. The most specific capability convolves a square fixed size neighborhood (2n+1 by 2n+1, typically with n=1,2 or 3) of each pixel with a kernel supplied by the user. Obviously, tools closer to the former are more desirable. Border processing options should be available, including constant-fill, reflection, and wrap-around (i.e. tiling). It should be optional to save results as individual images, or "pipe" them into other operators, as defined by an appropriate IP command language (oops, another solution method creeping in there...).

3.4) The method of 3.3 can be productively generalized to run along any defined geo-object. Performing neighborhood computations along the boundary of a region is a particularly useful operation for looking for gradient evidence in building models and model-matching routines.

3.5) Imagery algebra and arithmetic function operations are often performed between images; it should be easy to AND, OR or DIFF two images, and to do the operations +,-,* and / between them. Again, results should be savable as images.

3.6) A very efficient connected component capability should be available, as this routine is used often. It should be optionally 4 or 8 connected (hexagonal connectivity and "sided" connectivity options are nice too, but not as frequently used). Basic features should be computed for each component for efficiency (as they can be easy tracked during component construction, and are likely to be needed for further processing) including pixel-count, area in centimeters (if conversion constant is supplied with the imagery), number of pixels in the interior and exterior perimeters, average and variance of the gray-levels or for each of rgb, and genus (number of holes).

Assumptions: The data structures output by connected component are understood by virtually all other system components, including databases, display and browsing methods, and routines that process imagery features; see the level 2 description of step 4.

4) Extract features and measures from imagery regions.

4.1) Most features are calculated as functions using as input the numerical values and spatial relationships of a set of pixels in a connected region output by connected component. So the object-manipulating environment should make this data easy to obtain in arrays or other data structures for use in feature-computing functions.

4.2) The developer wants to create two basic classes of features. One is spatial structures, such as boundary shape descriptors and surface fits to a region, and the other is numerical (real-valued) measures of regions and their derived spatial representations, such as area, curvature, lengths, etc. Multi-resolution versions of most descriptors should be available. At any single level, the data structures representing the features at that level of resolution should be accessible by IP and feature creating routines the same way as single-resolution feature representations.

4.3) Whenever spatial representations are created, the developer wants to look at distributions of the associated measures, and to experiment with thresholding the distributions to look at various subsets of the feature space. The developer can specify the thresholding interactively as values, or can use one derived automatically from functions that look at distributions of feature values and execute criteria such as "threshold at top 5% of histogram of pixel values", of "threshold at the top 20% of high curvature points".

4.4) Search tools are used to define and process feature groupings. Feature search tools come in two basic varieties, attribute matching tools, and relational search. In attribute matching, set of regions or features are defined, typically as the output of some operator, and the developer wants to see which ones fulfill certain constraints on attributes, such as size, color, compactness, etc., stored with the feature or region. Relational search requires looking not just at each feature, but also at spatial relationships (and often other subsequent attribute comparisons) between multiple features or regions. These again come in two varieties, unordered relations, and sequential relations. Nearness is an unordered relation, but branching is an ordered, or sequential, relation. Boundary following is also a common sequential search operation.

Assumptions: All search tools understand any tools used for rapid (multi-resolution) spatial indexing of features and/or imagery.

4.5) Database storage of features is an implicit solution approach here, however if this approach is used, care must be taken to tightly integrate database access with IP and feature searching/generating routines. It may, for example, be far more efficient to pipe results from one operator to another without intermediate database storage. This could require some smarts, or option switches, to know when storage is desired. In any case, the user must understand what results are saved, which are not saved, and those that are not saved but whose process-to-create is recoverable.

5) Create models of objects to be recognized and measured from imagery.

5.1) The basic need is interactive modeling packages that can be used to create parametrized geometric object models from 1D and 2D splines, prisms, and generalized cylinders. In one scenario, a developer draws a contour on an image, extracts it and fits a1D spline to it. Another contour is extracted and spline fit; but now the ratio of of spline coefficients needs to be computed and stored, rather than forcing the absolute numerical parameters in. Relational model parametrization is of key importance; it is the ratios between parameters that get specified during the interactive sessions. Statistics governing distributions of parameters can be interactively input, but may come from other routines (e.g. a database access of a ground truth file of imagery or feature objects).

5.2) 3D models can be represented as an ordered, linked stack of 2D models, or directly as a 3D volume, as with generalized cylinders, or, with loss of information, as a surface map of polygons in 3-space with attachments. (Of course there are other representations, but these are the three we are initially using in medical applications.) In each representation the developer wants to view (projected) instantiated models against data interactively, and to view displays of models as parameters are interactively varied.

5.3) Geometric modeling proceeds by defining the primitive model components and their spatial relationships, such as adjacency, affixments, joints, and parametric relationships between axes and surfaces. It is convenient to have a modeling language that allows specification of spatial operations as an algebra (another design note).

5.4) Constraints are now modeled between geometric parts. Displays of tables of numerical outputs and also of projected models are used to check constraint propagation between model components are parameters are varied. For example, a developer models the bones of the finger and their relationships and constraints, and then checks the relative joint flexions by varying the angle of one joint and viewing displays of the range of the other.

5.5) Population statistics may be presented as normal distributions and/or by intervals. Constraints also can exist between these, so that when one distribution is pegged at a fixed value (or small interval) based on observations, dependent distributions are modified to ranges compatible with the observation and the relationship with the first distribution. Statistics may need to be computed from a training set, typically intended to be a random sample of the population being modeled.

Assumptions: The environmental tools either are sufficient to access the training set data, or the developer has some capability to access that data.

5.6) Now that the primitive model structures are understood, full part-of and is-a hierarchical (inheritance) taxonomies are defined, and the complete structure is created.

5.7) Operators are attached to the appropriate model nodes indicating parametric relationships between the operator inputs and the model so that the operator can be machine-instantiated at runtime to gather evidence supporting or denying the presence of an instance of the model.

6) Develop matching operators that compare regions, features and measures with segments of models.

6.1) The developer experiments with interactive parameter adjustment of models, projection of 3D models into 2D predictions, and matching the predicted model against extracted features and/or regions. There are two main matching approaches. The first is identical to model instantiation: an extracted feature is fit to a model component, e.g. a set of boundary pixels are fit to a 1D spline. In the second, a sensor image acquisition model is applied to an (partly) instantiated 3D model from a hypothesized perspective, and a geo-object is created that can be matched against the geo-object implied by the region occupied by the imagery feature. This fitting procedure can be supported by an appropriately applied least-squares-fit.

Assumptions: For each model primitive, there exists (a) method(s) to match it against some type(s) of features and/or regions.

6.2) A key advantage of model-based reasoning is that constraints in instantiated parts of models can be propagated to as yet uninstantiated values of model parameters to focus predictions for further processing. Based on partial matches, the developer now exercises the prediction mechanism to see the object localization implied by the partial match. This can be used to place values on operators, and as a guide to refining models and model matching.

Assumptions: Computation and display of predictions is easy and relatively rapid, and/or the development environment is multi-processing so that development does not have to stop and wait long for prediction results.

7) Develop inference structures such as Bayesian networks or rule bases.

7.1a) Building of the Bayes net begins by choosing the set of models that encompass the recognizable world (closed world assumption, with "other" category), and constructing a model bayes net that associates model nodes together as the competing hypotheses in Bayes nodes. (In the most general environment, there are cleverly indexed databases that allow automatic Bayes net building by building databases of conflicting models based on domain task applications.)

7.2a) Develop the conditional probability matrices between Bayes nodes based on the discriminatory evidence about the IP and other evidence gathering operators as described in step (5.7).

7.3a) Establish an initialization process that instantiates a (partial) Bayes net based on a fixed set of operations that have high probability of success (e.g. the medical hand-finder).

7.1b) Create a rulebase that captures model instantiations, evidence gathering, and decision making (about termination, for example), based on relationships between models and evidential matching results. The basic rule is of the form "If you see X, then do Y.".

Assumptions: Rule syntax and inference engines accept as inputs the results generated from evidence gathering operators, from statistical routines, and from accessing the model objects.

7.2b) Iteratively experiment with chaining in the rulebase based on alternate initialization sequences to determine both the initialization processing, and completeness of the rulebase.

8) Experiment with reasoning control strategies on the inference structures.

8.1) The developer iteratively changes the state of the Bayes net and/or rulebase or other inference structure, then manually indicates the next processing step and views the results.

Assumptions: Persistent data that the developer wants to save and reload to experiment with reasoning control includes all models, results of IP, pattern recognition, and grouping operators, matching methods, and Bayes nets and/or rulebases. It is preferable to be able to save an intermediate state in Bayes net and/or rulebase inference.

8.2) The developer runs automated processing routines such as a decision theory evaluation routine, a metarule selection strategy for multiple rule firings, or an influence diagram algorithm (that incorporates the Bayes net.)

Assumptions: Reasoning control programs accept as inputs models, inference structures and associated parameters.

8.3) The developer examines explanations of automated processing runs, and then interactively alters models, inference and/or reasoning control.

Assumptions: Explanation facilities are available for each control strategy and inference structure.

9) Craft the user interface and documentation to yield a natural vertical application solution.

9.1) Do task analysis, work studies and interface storyboards to determine the full sequence of steps, relations between domain task operations, environmental and operational constraints, resource constraints, required integration with other products and/or data, domain user technology familiarities, typical enduser occupational background and computer use capabilities (or phobias), and the uses that output information is put to.

Assumptions: The developer has access to view and analyze operational environments in the domain application area.

9.2) Rapidly prototype alternate user interfaces, and let potential end users experiment with the interface to uncover problems, design shortfalls, unexpected data dependencies, etc.

Assumptions: The developer has access to candid, knowledgable, representative and hardworking end users who have the time to evaluate the proto-interfaces.

9.3) Documentation for system is developed and tested on typical users both with and without assistance. Without assistance results are used to modify the written documentation so that assistance is unnecessary to easily run the system. With assistance results are used to test the successfulness of the actual system functioning under control of an enduser.

10) Test the (semi-) automated solution for robustness and reliability.

10.1) Determine approximately how many cases are required to establish reliability in the accuracy of the product's output measurements, then run the system over that many cases selected randomly from the population. Statistics need to be gathered.

10.2) Test that each system component has correctly implemented the required algorithms.

10.3) Establish software metrics for system reliability in terms of continuous functioning, and test the system accordingly. Again statistics are gathered, including timing and memory usage.

9.0 References

[Edelson et.al., 88] Edelson, D., J. Dye, T. Esselman, M. Black, and C. McConnell, "VIEW Programmer's Manual", Advanced Decision Systems, Mountain View, California, June 1988.

[KBVision, 87] "KBVision Programmer's Manual", Amerinex Artificial Intelligence, Amherst, Mass., 1987.

[Lawton and Levitt, 89] Lawton D.T. and T.S. Levitt, "Knowledge-Based Vision for Terrestrial Robots", Proc. DARPA IU Workshop, Morgan Kauffman, San Mateo, California, May, 1989. pp 128-133.

[Lawton and McConnell, 88] Lawton, D.T. and C.C. McConnell, "Image Understanding Environments", Proc. IEEE, Vol. 76, No. 8, August, 1988, pp. 1036-1050.

[McConnell et. al., 88] McConnell, C., K.Riley, and D. Lawton, "Powervision Manual", Advanced Decision Systems, Mountain View, California, 1988.

[Quam, 84] Quam, L., "The Image Calc Vision System", Stanford Research Institute, Menlo Park, California, 1984.

[Waltzman, 90] Unpublished notes at the Image Understanding Environment Requirements Meeting, Hughes AI Laboratory, Malibu, California, May, 1990.

Appendix A Object Hierarchy

```
Container

    Point
        Point1d
        Point2d
        Point3d
        PointNd

    Curve
        Curve1d
            Signal
        Curve2d
            SimpleCurve2d
                PointCurve2d
                EdgeCurve2d
                PolynomialCurve2d
                SplineCurve2d
                    BezierCurve2d
        Curve3d
            SimpleCurve3d
                PointCurve3d
                EdgeCurve3d
                PolynomialCurve3d
                SplineCurve3d
                    BezierCurve3d
        CurveNd
            SimpleCurveNd
                PointCurveNd
                EdgeCurveNd
                PolynomialCurveNd
                SplineCurveNd
                    BezierCurveNd


    Surface
        Surface2d
            SimpleSurface2d
                ConstantSurface2d
                    Box
                    Polygon
                        Parallelogram
                    RLE
                ValuedSurface2d
                    Image
                    PolygonImage
                        WarpedImage
                        TiltedImage
                    RLE_Image
        ConnectedSurface2d
            AggregateSurface2d
```

- Surface3d
    - SimpleSurface3d
        - ConstantSurface3d
        - ValuedSurface3d
    - ConnectedSurface3d
    - AggregateSurface3d
- SurfaceNd
    - SimpleSurfaceNd
        - ConstantSurfaceNd
        - ValuedSurfaceNd
    - ConnectedSurfaceNd
    - AggregateSurfaceNd

- Solid
    - Solid3d
        - SimpleSolid3d
            - ConstantSolid3d
                - GeneralizedCylinder
                - CSG's
            - ValuedSolid3d
                - Space
                - BoundedSpace
        - ConnectedSolid3d
        - AggregateSolid3d
    - SolidNd
        - SimpleSolidNd
            - ConstantSolidNd
            - ValuedSolidNd
        - ConnectedSolidNd
        - AggregateSolidNd

- HyperSolid
    - HyperSolidNd
        - SimpleHyperSolidNd
            - ConstantHyperSolidNd
            - ValuedHyperSolidNd
                - HyperSpace
                - BoundedHyperSpace
        - ConnectedHyperSolidNd
        - AggregateHyperSolidNd

- Coordinate
    - Global
    - Local
        - Base

- Collection (ContainerParts)

    - Array
        - ByteArray
        - Array2d
            - ByteArray2d

```
        Array3d
                ByteArray3d
        ArrayNd
                ByteArrayNd

Stream
        ByteStream
        Stream2d
                ByteStream2d
        Stream3d
                ByteStream3d
        StreamNd
                ByteStreamNd

Graph
        Tree
                List

Record
```

Appendix B Image Processing Source Libraries

ALV Toolkit

Contact: alv-users-request@uk.ac.bris.cs

Description:

Public domain image processing toolkit written by Phill Everson (everson@uk.ac.bris.cs). Supports the following:

- image display
- histogram display
- histogram equalization
- thresholding
- image printing
- image inversion
- linear convolution
- 27 programs, mostly data manipulation

BUZZ

Contact:	Tehnical: John Gilmore (404) 894-3560	Licensing: Patricia Altman (404) 894-3559

Artificial Intelligence Branch
Georgia Tech Research Institute
Georgia Institute of Technology
Atlanta, GA 30332

Description:

BUZZ is a comprehensive image processing system developed at Georgia Tech. Written in VAX FORTRAN (semi-ported to SUN FORTRAN), BUZZ includes algorithms for the following:

- image enhancement
- image segmentation
- feature extraction
- classification

HIPS

Contact: SharpImage Software
P.O. Box 373
Prince St. Station
NY, NY 10012

Michael Landy (212) 998-7857
landy@nyu.nyu.edu

Description:

HIPS consists of general UNIX pipes that implement image processing operators. They can be chained together to implement more complex operators. Each image stores history of transformations applied. HIPS is available, along with source code, for a $3000 one-time license fee.

HIPS supports the following:
- simple image transformations
- filtering
- convolution
- Fourier and other transforms
- edge detection and line drawing manipulation
- image compression and transmission
- noise generation
- image pyramids
- image statistics
- library of convolution masks
- 150 programs in all

LABO IMAGE

Contact:	Thierry Pun	Alain Jacot-Descombes
	+(4122) 87 65 82	+(4122) 87 65 84
	pun@cui.unige.ch	jacot@cuisun.unige.ch

Computer Science Center
University of Geneva
12 rue du Lac
CH-1207
Geneva, Switzerland

Description:

Interactive window based software for image processing and analysis. Written in C. Source code available. Unavailable for use in
for-profit endeavours. Supports the following:

- image I/O
- image display
- color table manipulations
- elementary interactive operations:
  - region outlining
  - statistics
  - histogram computation

- elementary operations:
    - histogramming
    - conversions
    - arithmetic
    - images and noise generation
- interpolation: rotation/scaling/translation
- preprocessing: background subtraction, filters, etc;
- convolution/correlation with masks, image; padding
- edge extractions
- region segmentation
- transforms: Fourier, Haar, etc.
- binary mathematical morphology, some grey-level morphology
- expert-system for novice users
- macro definitions, save and replay

Support for storage to disk of the following:
- images
- vectors (histograms, luts)
- graphs
- strings

NASA IP Packages

VICAR
ELAS -- Earth Resources Laboratory Applications Software
LAS -- Land Analysis System

Contact: COSMIC (NASA Facility at Georgia Tech)
Computer Center
112 Barrow Hall
University of Georgia
Athens, GA 30601
(404) 542-3265

Description:

VICAR, ELAS, and LAS are all image processing packages available from COSMIC, a NASA center associated with Georgia Tech. COSMIC makes reusable code available for a nominal license fee (i.e. $3000 for a 10 year VICAR license).

VICAR is an image processing package written in FORTRAN with the following capability:

- image generation
- point operations
- algebraic operations
- local operations
- image measurement
- annotation and display

- geometric transformation
- rotation and magnification
- image combination
- map projection
- correlation and convolution
- fourier transforms
- stereometry programs

"ELAS was originally developed to process Landsat satellite data, ELAS has been modified over the years to handle a broad range of digital images, and is now finding widespread application in the medical imaging field ... available for the DEC VAX, the CONCURRENT, and for the UNIX environment." -- from NASA Tech Briefs, Dec. 89

"... LAS provides a flexible framework for algorithm development and the processing and analysis of image data. Over 500,000 lines of code enable image repair, clustering, classification, film processing,
geometric registration, radiometric correction, and manipulation of
image statistics." -- from NASA Tech Briefs, Dec. 89

OBVIUS

Contact: for ftp --> whitechapel.media.mit.edu
otherwise --> heeger@media-lab.media.mit.edu
MIT Media Lab Vision Science Group
(617) 253-0611

Description:

OBVIUS is an object-oriented visual programming language with somesupport for imaging operations. It is public domain CLOS/LISP
software. It supports a flexible user interface for working with
images. It provides a library of image processing routines:

- point operations
- image statistics
- convolutions
- fourier transforms

POPI (DIGITAL DARKROOM)

Contact: Rich Burridge
richb@sunaus.sun.oz.AU
-- or --
available for anonymous ftp from ads.com
(pub/VISION-LIST-BACKISSUES/SYSTEMS)

Description:

Popi was originally written by Gerard J. Holzmann - AT&T Bell Labs.

This version is based on the code in his Prentice Hall book, "Beyond Photography - the digital darkroom," ISBN 0-13-074410-7, which is copyright (c) 1988 by Bell Telephone Laboratories, Inc.

VIEW (Lawrence Livermore National Laboratory)

Contact: Fran Karmatz
Lawrence Livermore National Laboratory
P.O. Box 5504
Livermore, CA 94550
(415) 422-6578

Description:

Window-based image-processing package with on-line help and user manual. Multidimensional (2 and 3d) processing operations include:

- image display and enhancement
- pseudocolor
- point and neighborhood operations
- digital filtering
- fft
- simulation operations
- database management
- sequence and macro processing

Written in C and FORTRAN, source code included. Handles multiple dimensions and data types. Available on Vax, Sun 3, and MacII.

ADVANCED DECISION SYSTEMS
a division of BOOZ • ALLEN & HAMILTON INC.
1500 PLYMOUTH STREET • MOUNTAIN VIEW, CALIFORNIA 94043-1230 • TELEPHONE (415) 960-7300 • FAX: (415) 960-7500

TR-09004-244-001-01

An Image Understanding Environment for DARPA Supported Research and Applications

Annual Technical Report
Draft

May 26, 1992

Contract Number: DACA76-89-C-0023
CDRL Sequence Number: A003
Period Covered: September 26, 1990 — September 25, 1991
ADS/BA&H Project Number: 09004-244-001

Advanced Decision Systems, a division of Booz·Allen & Hamilton Inc.
Douglas Morgan
Georgia Institute of Technology
Daryl T. Lawton

Prepared for:
Defense Advanced Research Projects Agency (DARPA)
and
U.S. Army Topographic Engineering Center
Telegraph and Leaf Roads, Cude Building 2592
ATTN: CEETL-RI
Fort Belvoir, VA 22060-5546

Abstract

ADS is participating in the design of a DARPA Image Understanding Environment (IUE). This report describes our contributions to designing the base classes, the structure of the class hierarchy and the user interface. The design benefits the IUE by allowing:

- Integration of the diverse concepts of IU within one environment.
- Rapid introduction of new users to the IUE.
- Organized extension of the base IUE by developers (including new users).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Section 1

INTRODUCTION

This is the annual technical report on work performed by the Advanced Decision Systems (ADS) Division of Booz-Allen & Hamilton (BAH) on the "An Image Understanding Environment for DARPA Supported Research and Applications" project DACA76-89-C-0023 during the period of September 26, 1990 to September 25, 1991.

The focus of activity during this period has been helping to design the DARPA sponsored Image Understanding Environment (IUE). ADS and both its subcontractors, Georgia Institute of Technology (GT) and Stanford University, are participating in this DARPA Image Understanding (IU) community design effort. The principal design participants have been Mr. Douglas Morgan (ADS), Dr. Tod Levitt (ADS, through September 1991), Dr. Daryl Lawton (GT), and Dr. Thomas Binford (Stanford).

The IUE will significantly advance the state-of-the-art in environments and development frameworks. Central to this advancement is the capability to seamlessly integrate the numerous concepts of IU into one system, to easily introduce new users to the system, and to extend the system in multiple new directions. Our work has focused on providing these capabilities with a class hierarchy clearly embodying IU concepts and a user interface allowing complex interactions to be simply expressed.

During previous reporting periods, the ADS/GT team designed a Vision Environment system and developed a prototype for a Sun workstation in C++. Plans were made for significant improvements to the ADS Vision Environment system prototype. However, the current reporting period began with a large cut in contract funding, leaving only enough support for scaled back design activities. At that time it was clear that further design activities would achieve the greatest impact if oriented toward influencing the emerging DARPA IUE design. Transitioning the Vision Environments design to the IUE and then continuing to refine the design would lead to the tangible result of a better IUE system end product. Like the Vision Environment system, the DARPA IUE is intended to facilitate the transfer of technology from the DARPA IU community into industrial, military, and commercial applications.

The IUE design is primarily being designed by a group of ten DARPA IU community representatives from industry and academia. The design group consists of:

Joe Mundy (Chair)	– GE
Thomas Binford	– Stanford
Terry Boult	– Columbia
Al Hanson	– U Mass
Bob Haralick	– U of Washington
Charlie Kohl	– AAI
Daryl Lawton	– Georgia Tech
Douglas Morgan	– ADS
Keith Price	– USC
Tom Strat	– SRI.

The design efforts by ADS, GT, and Stanford representatives have been funded largely through this contract. Prior to the formation of the design group, several meetings (open to the DARPA IU community) were held to refine the IUE goals and development strategy. During this period, Tod Levitt was instrumental in transitioning the Vision Environments design and design concepts into the IUE and in shaping the direction of the IUE design to initially focus on specification of a comprehensive class hierarchy. Presentations were also given by Tod Levitt and Daryl Lawton at the 1990 IU Workshop. One presentation reviewed the Vision Environments design and prototype development. The other described the functionality that a modern IUE should provide. Rand Waltzman and Oscar Firschein have been the DARPA program managers for the IUE effort.

This document describes the inputs of ADS team to the IUE design. These inputs were primarily in the definition of portions of a class hierarchy for the IUE. ADS was primarily responsible for Object Abstraction (the root classes of the hierarchy). GT was responsible for User Interfaces. Stanford provided support for both areas and additional inputs for Image Features. ADS also has acted as a liaison between the IUE design committee and the DARPA Open Object-Oriented Database project being carried out by Texas Instruments. All three organizations, with GT taking the lead, have helped refine the initial concepts for general spatial objects.

This document contains four sections. Section 2 provides background for the design goals and choices of the last two sections. Section 3 describes the Object Abstraction inputs to the IUE design. Section 4 describes the User Interface inputs.

Section 2

Background and Design Principles

The inputs of the ADS team to the IUE design are based on extensive experience in applying object-oriented techniques to IU. This chapter briefly describes some of the systems with which we gained this experience and presents several high-level design principles we have found to be critical to success.

The ADS inputs to the IUE design are based on experience in developing several generations of Object-Oriented class hierarchies and systems for Image Understanding (IU). These systems include:

- **Vision Environments.** This system was designed and prototyped during the first year of this contract. It was aimed at achieving high portability and cost effectiveness through use of C++ and off-the-shelf class libraries and tools. The design contained a early versions of spatial objects. The prototype integrated the Interviews class library with image handling, image display, and relational database access.

- **Sensor Algorithm Research Expert System (SARES) Testbed—CLOS Version.** This system has been under development with Wright Laboratories and DARPA since 1986. It integrates 3D objects, imagery, feature extraction, interpretation networks, Bayesian Networks, stochastic coordinate transformations, a multiple hypothesis (multiple worlds) mechanism, rule-based control, user interface (with image display, rendering, graphic overlay, network display, tables, and plots). This system addresses requirements, spanning: IU, environments, physical modeling, sensors, estimation, detection, classification, inference, user interfaces, and information management. Beginning about two years ago, ADS began a major reorganization of the Testbed to unify the approach of providing this wide range of capabilities. Since then, we have made major revisions to the testbed to make it reflect a more integrated view. The experience gained here has been a significant factor in the ADS inputs to the IUE design.

- **Sensor Algorithm Research Expert System (SARES) Testbed— C++ Version.** An effort is currently underway to convert a portion of the CLOS SARES Testbed to C++ .

- **View.** View, initially created in 1988, is an extensible class hierarchy (with efficient iteration macro environments, or "IU constructs") for image and image feature objects. It is an outgrowth of Powervision (see below), made with the goals of porting Powervision from ZetaLisp to CommonLISP and of separating the IU processing aspects of Powervision from the interactive user environment aspects. It supplies constructs for efficient iteration over all its spatial objects (images, edges, edges-lists, regions, etc.). Having been used on more than ten ADS projects, including SARES, View has influenced our contributions to image designs and efficiency considerations.

- **Powervision.** Developed and extended by Daryl Lawton while at ADS, Powervision is one of the first object-oriented IU environments. It is written in ZetaLisp/Flavors for the Symbolics Lisp Machine. A distinctive feature of this system is the extensive use of databases for library functions, processing results, and processing history.

- **ADS IUlab.** This is a system of abstract data types (ADTs) in C for types such as Image, Region, ConnectedComponent, ContainmentTree, and Table. This 1985 system implements strict ADTs in C, documenting the few nonfunctional (side-effect or procedural) access paths to object internals. It supports all C primitive numerical types (e.g. unsigned short and double) with typing macros. It supports different boundary handling techniques: reflection, constant value, and border value. To address memory constraints of early VAX systems, an Image can be read/created in any of three modes for disk buffering: all in-memory, read rows from disk on demand (with iterators making transparent access to disk), and read rows from disk on demand with back-up to a temporary disk file. This system also influenced our contributions on efficiency considerations.

With each system experience was gained in development, performance, training, portability and maintenance. Our proposals for the IUE design have been aimed at using the design principles distilled from these developments to arrive at a widely useful IUE. The issues in class hierarchy design that we considered include:

- **Aggregating names for similar concepts.** To ease training and maintenance, a system should consistently name implementations of similar concepts. An example from CommonLISP where naming is not consistent is the set of access functions for the various attribute/value data structures. These access functions include: aref, assoc, elt, get, getf, gethash, nth, nthcdr, rassoc, and slot-value. Although these all do the same conceptual operation (evaluate an index/value association), there are functions with ten different names and the order of arguments changes unpredictably between functions. Without conscious steps to aggregate names, the IUE could easily generate many dozens of more names for the same operation. A tightly knit class hierarchy is an excellent way to encourage consistent naming in the IUE.

- **Specifying precise semantics.** To make it possible to do up-front design and to write comprehensive test suites, the class documentation needs to precisely define the over-time and between-object behaviors of objects. It is especially important to document what to expect for identity, creation, destruction, copying, assignment, equality, persistence, and mutation of objects. Consistency of these aspects of object behavior will greatly affect the ability to integrate contributions from a distributed set of developers.

- **Taking a set- and relation-theoretic approach.** We find that sets, relations (tuple sets), functions (functional relations), and networks (sets of relations) are fundamental to many aspects of the SARES Testbed and our other IU environments. These aspects include:

  - Databases (set of relations)
  - Bayesian Networks (set of conditional probability relations)
  - User interface (sets of object-display-action relations)
  - Coordinate system networks (sets of coordinate transform functions)
  - 3D models (sets of attachment relations)
  - 3D primitives (sets of points)

  By capturing this commonality in a single set-theoretic class hierarchy, names for concepts are tightly aggregated and much of the system's semantics can be specified for just one class and reused many times without change. A single narrow-rooted class hierarchy also allows the objects of a strongly typed language (such as C++) to participate in a wide range of activities defined along the trunk of abstract set-theoretic classes.

Although we had earlier recommended that C++ be the sole delivery language for the Vision Environment system, for the IUE a mix of CommonLISP and C++ is a requirement. CommonLISP adds the significant advantages of automatic garbage collection, dynamic compilation, dynamic loading, interpreted operation, and extensive support for multiple inheritance. These factors, in addition to the reliance of many DARPA IU contractors on CommonLISP, will help with timely construction of robust software for complex IU applications.

Section 3

Object Abstraction

This chapter presents the Object Abstraction inputs to the IUE design. These inputs have been incorporated into the IUE design document produced by the IUE design committee and into the IUE Overview paper presented at the 1992 DARPA IU Workshop.

3.1 Overview

Object Abstraction for the IUE lays out the trunk of the class hierarchy and proposes software development guidelines. Its aim is to smooth the way for independently developed IUE components to work together. The IUE class hierarchy is organized around a single root class (*Object*), a metaclass (*Class*), and a core set of classical mathematics, physics, and information processing related classes. Guidelines for extending the class hierarchy are also included.

The IUE will be a large and extensible system jointly developed at numerous sites throughout the country. Bringing order to the IUE build will require one specific class hierarchy and one specific set of design principles. This will provide coordination well beyond the generic practice of "object-oriented programming" or "OO design". The hierarchy and design principles of the IUE must support the built-in IUE capabilities and accommodate new additions. A good foundation will keep the overall costs of building the IUE down and will improve chances for a successful, robust system.

Object Abstraction aims to help developers provide consistent capabilities and names for classes and methods. It also aims to ensure that capabilities are comprehensive and logically arranged with well-defined paths to obtaining maximal efficiency. The overall effect is to allow each new development to be added to the class hierarchy at the logically correct point (rather than, for example, making a new incompatible class hierarchy or work with none at all).

Object Abstraction is essentially one *Object* class, one *Class* class, a *Collection* class hierarchy, and design/programming guidelines for the IUE build. These foundation

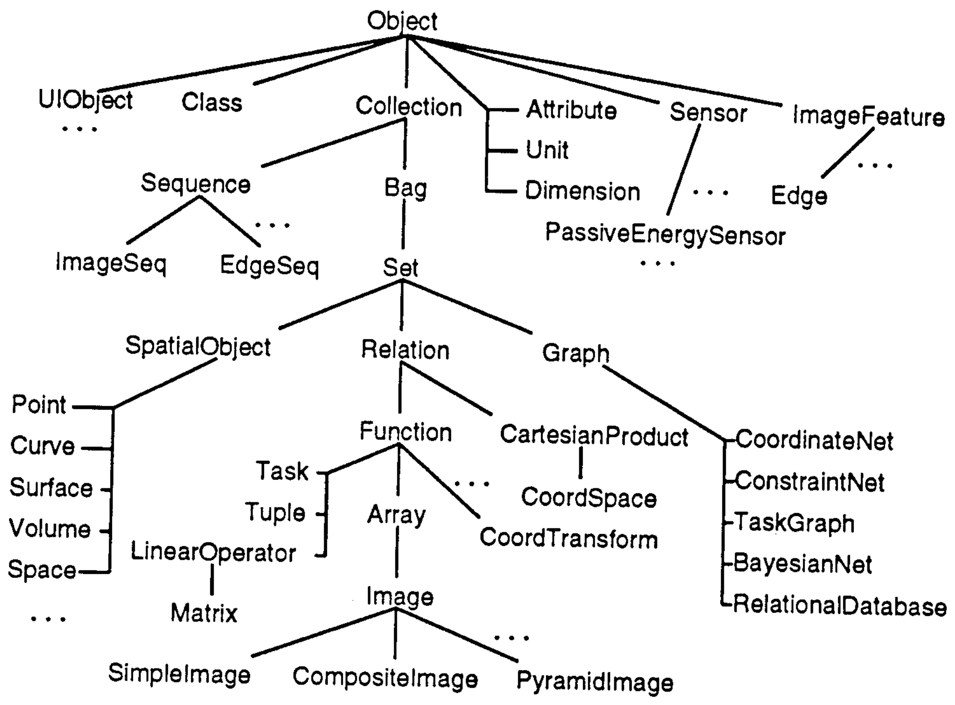


Figure 3-1: Object Hierarchy Outline.

classes define an extensive set of methods for interacting with environment-level tools, for tailoring specialized classes via parametric hooks, and for consistently accessing common mathematics and physics operations. Figure 3.1 outlines the class hierarchy. The figure is an outline of abstract classes: the IUE will have more classes to provide various implementations of the classes shown in the figure. Also, aspects such as name choice, specific inheritance paths, and the use of multiple inheritance are being further refined.

3.1.1 *Object* and *Class*

Together, *Object* and *Class* define the environment-level behavior shared by all objects. All IUE classes inherit from *Object* and are associated with a unique instance of the class *Class*. These classes allow the environment to examine instances and configure operations (especially I/O, copy, and display/editing) based on different types and classes. This two-class approach is closely aligned with systems such as CLOS, Smalltalk, and the National Institute of Health C++ Class Library. Applying the approach to C++ requires substantial infrastructure and discipline, but will be valuable in organizing the IUE and providing powerful interactive capabilities.

3.1.2 Math, Physics, and Information Processing Classes

The math, physics, and information processing classes represent such fundamental concepts as images, extracted features, world objects, pure geometry, transformations (including coordinate systems), sensors, sets, sequences, relations, and networks of relations. The *Collection* class is basic to much of the class hierarchy.

Collection is parameterized with functions (including equivalence, insert compatibility, and union compatibility) so that it can cleanly specialize in multiple directions:

- Finite, countably infinite, and uncountable (at least conceptually so) numbers of elements
- *Object*-valued or language-primitive-valued (e.g., int) element types
- Constraints on types or values of elements to be inserted.

Early versions of the IUE design had nearly every class branching of the trunk on Figure 3.1 inheriting directly from *Object*. The current designs enforce much greater uniformity among important mathematics and physics classes by moving them to more meaningful positions further down the class hierarchy.

3.1.3 Development Guidelines

The development guidelines of Object Abstraction include:

- A dictionary of translating between terminology for C++ , CommonLisp, and standard OOP.
- Naming conventions for methods — these specify such characteristics as return type, inlining, image boundary handling, immutable versions, etc.
- Notions of abstract type and implementation hierarchies embedded in the class hierarchy — this separates method definitions from slot definitions so that highly constrained objects far down in an inheritance hierarchy can have the union of all supertype methods names without carrying around extra slots defined in many unrelated implementations of supertypes.
- Use of source preprocessing tools to aid C++ development.
- Use of persistent object techniques.
- Views of objects — views objects wrap around other instances to change the apparent set of methods or interface (e.g., a view is a low-cost and consistent way of creating a vector-valued image from a sequence of images and visa versa).

The guidelines aim at producing efficient class hierarchies free of semantic conflicts on methods or slots (i.e., potential conflicts due to multiple inheritance are not left chance).

3.2 Classes and Conventions

This section shows how the IUE class hierarchy is organized around a single root class (*Object*), a single metaclass (*Class*), and a variety of math and physics oriented classes. *Object* and *Class* work together to define the environment-level behavior shared by all objects. The math and physics classes define the fundamentals of world objects, relations, sensing, networks, and image understanding. Classes for I/O channels and display/interaction devices are not covered.

To help with the organization of a powerful and uniform environment, we propose effective approaches for several areas, including:

- Vocabulary for the object system (especially since we use both C++ and CLOS in the IUE)
- Capabilities that all objects share
- Tools to mechanically generate code required by the environment (such as for class object creation, instance copying, and I/O)
- Naming conventions (especially with regard to how names map into combinations of auxiliary concepts such as inlining, declaring the return type, dynamic binding, static binding and function name overloading, mutable versus immutable objects, free versus member functions)
- Hierarchy that introduces math and physics related methods early on in a few central classes (e.g., `hasMember` for a collection can be used many levels down in the hierarchy to test whether a point is inside 3D object). The hierarchy defines the primary interfaces for each object type (declaring, for example, that an *Image* has primarily a functional interface rather than that of a kind of 3D object with pose.)
- Parametric freedoms allowing the *Collection* class to be specialized to such diverse objects as sequences of arbitrary *Objects*, `unsigned char` valued arrays, and procedurally defined *Functions*
- Network classes sufficient for managing constraints, relational databases, and Bayesian inference networks
- Abstract classes for separating method interfaces defined for a class from the internal representations used by instances. This is needed so that highly constrained objects far down in an inheritance hierarchy do not have to contain the union of slots for all the complex, less constrained objects above.

- View objects for resolving method name conflicts that arise when two logical views of one instance would naturally use the same method name, but for different purposes.

Each of the above issues is now addressed.

3.2.1 Vocabulary

Table 3-1 lists C++ and CLOS names for approximately equivalent concepts. The table also lists our preferred usage in the left column.

3.2.2 Common Capabilities

The capabilities common to all classes are organized around a single root superclass and a metaclass (a class whose instances each describe a unique class). The approach using these two classes is very close to that followed by CLOS, the C++ National Institute of Health Class Library (NIHCL) and Smalltalk. Applying the approach to C++ requires considerable discipline. For CLOS, it is relatively easy.

We base all IUE classes on a single root superclass, *Object*. There are at least two strong arguments doing so. First, in OOP, this is a natural way to be sure that every instance has at least the minimum interface to interact with the standard tools of the environment. Second, without a root *Object* class, the strong typing of C++, forces the use of parameterized versions of subclasses of *Collection* to store instances from each disjoint inheritance hierarchy. Storing instances from disjoint inheritance networks in one collection requires type-violating casts.

We have also chosen to define one class (a metaclass) whose instances represent the characteristics of some other class. An environment often needs to examine the structure of an object and choose operations based on different types and classes. As C++ provides no default way to query an instance about its class, we define one.

3.2.3 Code Generation

It is critical to make creating a new class easy. In C++, this almost certainly implies tools to scan *.h files and mechanically generate code for such activities as creating a class object, and defining the virtual functions supporting instance copying, I/O, display, editing, etc. Although this section points out the need for such tools, it does not present their design.

Table 3-1: Vocabulary Choices for Object-Oriented Programming Concepts.

Preferred Names	Other Names and Usages
Virtual Function	Generic Function (with Dispatch on Only First Argument), Message, Virtual Member Function
Generic Function	Multimethod, Would be a Virtual Function on more than One Argument (This Restricted Notion of Generic Function is Nonstandard, but Makes Clear a Useful Distinction.)
Instance	Object
Class	No Other Name
Method	Virtual Function Definition
Call a Virtual Function	Send a message
Call a Generic Function	Method Dispatch
Self	First Argument of a Generic Function
Method Combination	Daemons (no built-in C++ analog)
Superclass	(Virtual) Base Class (C++ and CLOS Handle Conflicting or Ambiguous References to Slot and Methods Differently)
Direct Superclass	Direct Virtual Base Class
Indirect Superclass	Indirect Virtual Base Class
Subclass	Derived Class
Direct Subclass	Direct Derived Class
Indirect Subclass	Indirect Derived Class
Nonvirtual Base Class	No good CLOS Analog
Abstract Class	Mixin (Eiffel uses Deferred Class)
Nonvirtual Member Function	Statically Bound Member Function (CLOS does not directly support static overloading of function names. Also, CLOS on specializes on a fixed number of arguments for all methods)
Slot	Member Object, Instance-Allocated Slot (bInstance Variable
Static Member Object	Class-Allocated Slot (Smalltalk uses Class Variable)
Free Function	No Other Name
const Object	Immutable (no associated CLOS syntax)
inline Function	Macro Expanded or Inlined

3.2.4 Naming Conventions

We use several naming conventions in this section. The conventions (adapted from X-Windows standards) are:

- The main part of class names begin with a capital letter. Class names with multiple words are concatenated with each word capitalized. Sometimes modifiers, such as component datatype names, are appended to the main part after an underscore. An example class name is *SomeKindOfImage_int.*
- Slot names are all lower case and, if multiple words long, have words separated by single underscores. An example slot name is *this_is_a_slot.*
- Function names have the following format (bracketed items are optional):

  mainPart[*_ReturnType*] [*_PostfixModifier*]

- The return type of a function may be specified by name after the main part of a function name and before the postfix modifiers.
- Postfix modifiers of function names are (at least) of the form:

  `[_[[n]c][[n]i][[n]m][[n]v]]`

  where:

  o `n` - Not

  o `c` - Const (function does not modify first argument's value)

  o `i` - Inline

  o `m` - Member Function

  o `v` - Virtual

- The characters <T> can be freely replaced with the name (possibly using an encoding scheme) of any standard C++ or Lisp datatype (e.g., `unsigned_char` or `double`).

The postfix modifiers allow the developer to consistently name the multiple different functions (up to 16 for one *mainPart* name) that perform one logical task, but satisfy different requirements for speed, safety, time of binding, and ability (in C++) to reference functions by pointer. (Actually, the "member" and "not member" options are redundant since C++ allows both types of functions to be named identically and still be uniquely referenced using "::" syntax.) Other modifiers options can be added to indicate choices such as image access boundary handling (e.g., no checking, reflection, wrapping, default value).

The postfix modifiers have no defaults. That is, the exact meaning of an unadorned function varies from function to function. The designer is free to choose the unadorned name for the most commonly used version of the function. The `n` option can negate any of the `c`, `i`, `m`, or `v` options implicitly chosen for the unadorned function.

In C++ and CLOS, the return type of a function does not enter into the generic or virtual function method dispatch calculation. For efficiency, we often want to have logically similar functions be implemented completely differently depending upon desired output types. E.g., an inline method that returns a `char` from a `char` array object could be many orders of magnitude faster than a similar method that returns an instance whose state represents a `char`. The naming conventions address this need with the *ReturnType* portion of the function name.

The methods chosen to be defined in this section do not use postfix modifiers. Modifiers will be important to the final IUE, but are not essential to the presentation of an overall logical design.

3.2.5 Hierarchy

The class hierarchy (of Figure 3.1) includes the root *Object* class, the *Class* class, and math and physics related classes. The figure shows the hierarchy only to depths where it touches the top level classes being designed by other organizations represented on the IUE design committee. The hierarchy shows that an *Image* inherits from *Array*, *Function*, *Relation*, *Set*, *Bag*, *Collection*, and *Object*. This structure prescribes a host methods for *Image*. The definitions of many methods will be inherited, the remainder are supplied by the implementations of *Image* and its subclasses.

3.2.6 *Collection* Class

The *Collection* class is basic to much of the hierarchy. It must cleanly specialize in multiple directions. Different subclass need to represent

- Finite, countably infinite, and uncountable (at least conceptually so) numbers of elements
- *Object*-valued or language-primitive-valued (e.g., int) element types
- Constraints on types or values of elements to be inserted.

Parameterization by two functions makes *Collection* sufficiently flexible. These functions are the equivalence function and insert compatibility test. Additional useful parameterizations are a union compatibility test (actually derivable from the insert compatibility test) and a potential constraint on element type.

3.2.7 Network Classes

Relational databases, Bayesian networks, function composition networks, and constraint networks are all networks for managing joint information over a set of attributes. The structures necessary to properly support these networks extend well beyond the usual undirected and directed graphs of links and arcs connecting two nodes at a time. (We choose to use "link" rather than "edge," the standard, to avoid clashing with the image related usage of "edge.") Of particular importance are generalizations of links and arcs to connections between sets of nodes. These are the hyperlink, or set of nodes, and what we will call the hyperarc, or directed pair of sets of nodes. A set of hyperlinks forms a hypergraph and a set of hyperarcs forms what we will call a dihypergraph (directed hypergraph). Also necessary are attributes (nodes) with their domain sets. Further, a structure at the heart of efficient inference and query algorithms is the Join tree of hyperlinks (undirected tree of hyperlinks with the intersection of any two hyperlinks contained in every hyperlink in the connecting path).

3.2.8 Abstract Classes

In this description of the hierarchy, *Collection* and its subclasses are abstract classes. That is, classes that define method name and signatures, but do not provide slots and implementation. This allows objects of widely varying implementation but identical interface to be created with no superfluous per-instance overhead. As the IUE develops, the hierarchy will be filled in with multiple subclasses supplying specific implementations for each abstract interface.

3.2.9 Views

With a complex hierarchy built largely on sets, relations, and functions, it is inevitable that one logical view of some class will involve a set, relation, or function interface, but not in the way implied by the inheritance hierarchy. For example, the inheritance defined view of a *BayesNet* is as a directed hypergraph with set-like interface to a structured set of *Hyperarcs*. Because a *BayesNet* can logically be viewed as a joint probability density (a function), it could also have a set-like interface to a structured set of *Tuples* (mapping random variable values to real numbers). A *View* object would wrap around the *BayesNet* instance and convert the *Set* interface methods of union, insert, etc. to operate on the set of *Tuples* instead of the set of *Hyperarcs*. We propose that view creating methods (such as *asFunction*) be provided as needed to wrap instances with new instance with modified interfaces. The view objects of Interviews are similar to our *View* objects, but they are used only to redefine the user interface methods of an object instead of any set of methods. This section will not specify view objects other than to say that there will be many view subclasses and their use should be an integral part of the environment.

3.3 Hierarchy Class Definitions

3.3.1 Object

The *Object* class is the root superclass of all classes in the environment. In conjunction with the *Class* metaobjects, *Object* specifies the interface for the environment-level operations that apply across all instances. These operations include:

- I/O
- Display and hardcopy
- Type queries
- Equality queries (default equivalence function)
- Copying (multiple semantics)
- Mutability and locking control
- Environment queries (version, source code, object code, documentation)
- Inspecting and editing
- Memory management (to be specified)
- Persistence (to be specified).

Table 3-2 presents the method signatures (i.e., names, input types and output types) for *Object.* Table 3-3 expands the signatures to include documentation of each method.

3.3.2 Class

A *Class* describes an object class. Each class has a corresponding unique *Class* instance. Table 3-4 shows the method signatures and Table 3-5 shows the definitions.

Table 3-2: *Object* Method Signatures.

Name	Input Types	Output Types
newLike	()	Object
copy	()	Obj:Object
deepCopy	()	Obj:Object
shallowCopy	()	Object
deepenShallowCopy	(Object)	Object
classOf	()	Class
isSame	(Object)	Boolean
isKindOf	(Class)	Boolean
species	()	Boolean
isSpecies	(Object)	Boolean
isUnique	()	Boolean
isImmutable	()	Boolean
changeToImmutable	()	Status
isLocked	()	Status
lock	()	Status
unlock	()	Status
destroy	()	Status
deepDestroy	()	Status
reset	()	Status
isEqual	(Object)	Boolean
display	(Displayable)	Status
inspect	()	Status
edit	()	Status
describe	(OStream)	Status

Table 3-3: *Object* Methods Definitions

Name	Input Types	Output Types
newLike	()	Object
Make a new instance of the same class as self and load with default values.		
copy	()	Obj:Object
The "natural" copy. Make a new instance of the same class as self. Copy "shallowly" those slots that expect address equality or whose objects are immutable. Copy others "deeply." Further, *copy* is free to change the internal structure as long as self->isEqual(Obj) would be True if executed and value of a mutable object is not stored by copying the mutable object's address.		
deepCopy	()	Obj:Object
Make a new instance of the same class as self, copy non-pointer values from self to Obj, and set all internal pointers of Obj to deepCopy's of what self points to.		
shallowCopy	()	Object
Make a new instance of the same class as self, copy non-pointer values from self to Obj, and set all internal pointers of Obj to point where the internal pointers of self point.		
deepenShallowCopy	(Object)	Object
Turn a shallowCopy'ed instance into a deepCopy'ed one. Method used by deepCopy.		
classOf	()	Class
Return the Class instance corresponding to the class of self.		
isSame	(Obj:Object)	Boolean
Return True iff self and A are Obj are identically the same object (same address).		
isKindOf	(c:Class)	Boolean
Return True iff self is an instance of the class *c* or its subclasses.		
species	()	Boolean
Return the class (often abstract) in the hierarchy that defines the semantics of the state of self.		
isSpecies	(Obj:Object)	Boolean
Return True iff self and Obj are of the same species.		
isUnique	()	Boolean
Returns True iff self is the only object that can obtain its current state.		
isImmutable	()	Boolean
Returns True iff the apparent state of self can never (again) be changed with the object interfaces. Once self->isImmutable() returns True, it should always return True.		
changeToImmutable	()	Status
If possible, change the state of self so that self->isImmutable() will return True.		

Table 3-3: *Object* Methods Definitions (cont'd)

Name	Input Types	Output Types
isLocked	()	Status
Return True iff the apparent state of self cannot currently be changed with the object interfaces.		
lock	()	Status
If possible, change the state of self so that self->isLocked will return True.		
unlock	()	Status
If possible, change the state of self so that self->isLocked will return False.		
destroy	()	Status
Destroy self.		
deepDestroy	()	Status
Recursively destroy self and all its contained objects where what is "contained" is class-specific.		
reset	()	Status
Reset state of self to the class-specific default.		
isEqual	(Obj:Object)	Boolean
isEqual forms the species-specific "natural" equivalence relation between instances. It returns True iff self and Obj are "equal".		
display	(D:<Displayable>)	Status
Display self on *D*.		
inspect	()	Status
Initiate the interactive inspection of self.		
edit	()	Status
Initiate the interactive editing of self.		
describe	(Out:<OStream>)	Status
Describe self on Out.		
name	()	String
Return the name of the class referred to by self		
directSuperclasses	()	Set
Return the set of class instances of the direct superclasses of class referred to by self.		
superclasses	()	Set
Return the set of class instances of the superclasses of class referred to by self.		
directSubclasses	()	Set
Return the set of class instances of the direct subclasses of class referred to by self.		
subclasses	()	Set
Return the set of class instances of the subclasses of class referred to by self.		

Table 3-3: *Object* Methods Definitions (cont'd)

Name	Input Types	Output Types
members	()	Set
Return a set of slot and function descriptors describing the class referred to by self. A descriptor should include name, slot or function, type, signature (if describing a function), access restrictions, file name where function is defined, whether static or class-allocated, etc. Building descriptors requires parsing code (or perhaps using symbol table information).		
headerFile	()	FileName
Return the file defining the class referred to by self.		
version	()	Version
Returns a version number or name for the class referred to by self.		

Table 3-4: Method Signatures

Name	Input Types	Output Types
name	()	String
directSuperclasses	()	Set
superclasses	()	Set
directSubclasses	()	Set
subclasses	()	Set
members	()	Set
headerFile	()	FileName
version	()	Version

Table 3-5: Method Definitions

Name	Input Types	Output Types
name	()	String
Return the name of the class referred to by self		
directSuperclasses	()	Set
Return the set of class instances of the direct superclasses of class referred to by self.		
superclasses	()	Set
Return the set of class instances of the superclasses of class referred to by self.		
directSubclasses	()	Set
Return the set of class instances of the direct subclasses of class referred to by self.		
subclasses	()	Set
Return the set of class instances of the subclasses of class referred to by self.		
members	()	Set
Return a set of slot and function descriptors describing the class referred to by self. A descriptor should include name, slot or function, type, signature (if describing a function), access restrictions, file name where function is defined, whether static or class-allocated, etc. Building descriptors requires parsing code (or perhaps using symbol table information).		
headerFile	()	FileName
Return the file defining the class referred to by self.		
version	()	Version
Returns a version number or name for the class referred to by self.		

3.3.3 Collection

A *Collection* represents an object that "collects" or "contains" other objects. The *collection* class specializes to many classes including Bag, Set, Relation, Function, and Image. A raw *Collection* has axioms that allow for remembering duplicate elements and insert order. A *Bag* adds an axiom that makes it impossible to remember insert order. A *Set* adds an axiom that makes adding an element many time the same as adding it only once. Other subclasses add axioms to restrict the types of elements that can be added (e.g., *Tuples* with instance-specific schema for relations) and vary the notion of equivalence between elements (e.g., a *IdentSet* would check the address of instances while an IntSet would check for the numerical value of instances or primitive datatypes.) Table 3-6 presents the signatures of a minimal set of methods for *Collection.*

Elements must satisfy `insertCompatible` to be valid arguments to `insert`. Other collections must satisfy `unionCompatible` to be valid arguments to `union`, `intersect`, etc. Many subclasses of *Collection* do not have to to check for argument compatibility since any argument that passes static type checking (or dynamic dispatching) will be acceptable.

A fundamental property of each collection is its equivalence function (*equivalentArgs*) for determining if two objects are equal for all purposes of the collection. A collection is best thought of as containing abstract elements rather than objects themselves (only when the equivalence function is object equivalence not a more general function of internal state). An element should be thought of as naming the abstract object equivalent to the instance. For example, a collection does not necessarily "return" the same instances (as chunks of memory) that are inserted, it can return any equal (under the equivalence function) objects. Also, any two sequences of operations differing only by substitution of equivalent instances and ending in a "isMember" or "isSubset" type of test will yield identical test results.

Equivalence relations allow collections to be extensible and highly efficient. For instance, suppose that "object value" is the equivalence relation for a certain set. Then, the collection does not guarantee that an object, once inserted, can ever be recovered. The class simply guarantees that an object of the same value can be recovered. Specialized collection classes may be then able to use efficient memory schemes (like arrays) to minimize storage and to speed computation. It can also considerably reduce the work required of a persistent storage system. Only values needed to copy objects (not objects themselves) have to be stored, thus eliminating the overhead of maintaining persistent UID's. The option for storing memory-objects is always available by making the equivalence function be #'EQ (or &a == &b). Cartesian product equivalence functions are used for relations and functions. This unifies numerous data types often treated more independently. Examples include

Table 3-6: *Collection* Method Signatures.

Name	Input Types	Output Types
equivalentArgs	(Object, Object)	Boolean
unionCompatible	(Coll:Collection)	Boolean
unionCompatible	(Primitive:<T>)	Boolean
insertCompatible	(Obj:Object)	Boolean
insertCompatible	(Primitive:<T>)	Boolean
isSingleton	()	Boolean
isEmpty	()	Boolean
isFinite	()	Boolean
isCountable	()	Boolean
isCountablyInfinite	()	Boolean
isUnCountable	()	Boolean
cardinality	()	int
cardinality1	()	Cardinal
hasMember	(Obj:Object)	Boolean
hasMember	(Primitive:<T>)	Boolean
disjoint	(Coll:Collection)	Boolean
isSubset	(Coll:Collection)	Boolean
insert	(Obj:Object)	Collection
insert	(Primitive:<T>)	Collection
remove	(Obj:Object)	Collection
remove	(Primitive:<T>)	Collection
remove1	(Object)	Collection
remove1	(<T>)	Collection
union	(In:Collection)	Out:Collection
difference	(In:Collection)	Out:Collection
intersection	(Collection)	Collection
relativeComplement	(Collection)	Collection
symmetricDifference	(Collection)	Collection
choose1	(Collection)	(Object Boolean Collection)
choose1	(Collection)	(<T> Boolean Collection)
map	(Collection, func,Class)	Collection
map	(Collection, func,Class)	Collection
mapc	(Collection, func)	void
mapc	(Collection, func)	void

numerical arrays, object arrays, hashtables, relational tables, and procedurally defined functions.

If the equivalence function is object identity (which we will call the object eq), then the actual instance inserted into the collection has to be retrievable/testable. If the equivalence function allows different objects with the same "values" to be equal, then there is no guarantee that the instance inserted is actually stored (e.g., a relational table will generally disregard an inserted tuple object after the tuple's field values have been extracted and stored in a relation-specific internal form). An equal-collection cannot rely on storing a mutable object as the sole memory of an insert as the object could mutate and no longer denote the same value it did at time of insert.

Persistent collections introduce further considerations (e.g., a persistent eq-collection can contain only persistent objects). A set of consistency rules and implementation possibilities are as follows:

- persistent eq-collection $\Rightarrow$ elements must persist
- eq-collection $\Rightarrow$ can use any object reference (e.g., persistent Object ID (OID) or pointer) for representing elements in working storage
- equal-collection and immutable element $\Rightarrow$ can use any object reference or any equivalent data for representing the element in working storage
- equal-collection and mutable element $\Rightarrow$ element must be "copied" into immutable equivalent data for working and/or persistent storage
- persistent eq-collection $\Rightarrow$ must use OID for representing elements in persistent storage
- persistent equal-collection and immutable element $\Rightarrow$ can use OID (if available) or data for representing element in persistent storage.
- persistent equal-collection and mutable element element must be "copied" into immutable equivalent data for working and/or persistent storage

3.3.4 Bag

A *Bag* is a collection of order independent elements with meaningful duplicates. The method signatures are unchanged from those of *Collection.*

3.3.5 Set

A *Set* is a collection of order independent elements with duplicates meaning the same as only one. The method signatures are unchanged from those of *Collection.*

3.3.6 PointSet

A *PointSet* is set of points in an n-dimensional Euclidean space.

3.3.7 PointSpace

A *PointSpace* is an n-dimensional Euclidean space. Abstract points in PointSets can be arbitrarily subtracted to form vectors. Vectors can be added to points. Elements of base PointSets have no scalar multiplication operator and only barycentric addition (coefficients sum to one). Elements of base PointSets are abstract, without methods for coordinate representation, default point frame, or default vector basis. (They can be, for instance, abstract handles to locations in the real world.) The base PointSet class is specialized in different ways to get many important classes. Through different types of specialization, instances of subclasses may have:

- Specific embedding dimensions (dimension of the PointSpace containing the PointSet).
- Specific embedded dimensions (manifold dimensions of the PointSet itself).
- Points with associated coordinate representations (with respect to a distinguished frame per instance).
- Vectors for elements. There is a distinguished origin per instance, and new operators are defined. These include multiplication by a scalar, vector addition, inner product, and (for three-dimensional vectors) cross-product.
- An interpretation as a specific physical concepts (e.g., physical space, time, space-time, image pixel locations, window pixel location, viewport space, screen locations, color space, spline function space).

Such specializations include: Frame, Basis, Interval, Region, Volume, TimeInterval, DurationInterval, ScreenFrame, ColorBasis, TimeFrame, VectorSubspace, CoordinatePointSpace CoordinateVectorSpace, and TriangularFacet.

A potentially useful way to specify the class of points that are elements of a PointSet is to identify singleton sets with points. With this, operations automatically generalize from points to sets of points and there is no need to define classes both for sets of points and for points themselves. This meriology is the most common method of treating schemas in the database literature. Also, Wand has suggested it as part of a general approach to modeling real things.

A request for a change of basis (change of frame) includes the new basis (frame) for the coordinate representation. This is different from a transformation operation that keeps the same basis or frame (if one is used at all). Dynamic type checking should

Table 3-7: Additional *Relation* Method Signatures

Name	Input Types	Output Types
join	(Relation)	Relation
relationalDivide	(Relation)	Relation
select	(Expression)	Relation
select	(Function)	Relation
projectOn	(Set)	Relation
projectOff	(Set)	Relation
thetaJoin	(Relation)	Relation
semiThetaJoin	(Relation, Expression)	Relation
attributes	(Relation)	Set
addAttribute	(Relation, Attribute, Object)	Relation
removeAttribute	(Relation, Attribute)	Relation

not allow operations on mismatched frames (or, if possible, will automatically put two coordinate representation into the same frame via appropriate change of frame calculations).

3.3.8 Relation

A *Relation* is a set of tuples, all having the same domain (schema). A tuple is a mapping from an index set into values. The index set is a set of attribute objects, not simply a set of consecutive numbers starting at zero or one. Table 3-7 gives the additional method signatures for a *Relation.*

3.3.9 Function

A *Function* is a relation in which the values of the dom attribute set uniquely determine values of the range attribute set. Every point in a dom set maps to a point in the ran set . Further, every point in the ran set corresponds to one or more points in the dom set. dom and ran belong to (possibly larger sets) domain and range. Table 3-8 gives the additional method signatures for a *Function*

3.3.10 Tuple

A *Tuple* is a function with a domain that is a set of attributes.

Table 3-8: Additional *Function* Method Signatures

Name	Input Types	Output Types
domain	()	Set
range	()	Set
dom	()	Set
ran	()	Set
isPartial	()	Boolean
isOneToOne	()	Boolean
isOnto	()	Boolean
isInvertible	()	Boolean
evaluate	(Object)	Object
evaluate	(<T>, <T>, ...)	Object
atPut	(Object, Object)	Boolean
compose	(Function)	Function
composeClass	(Function)	Function
composeCompatible	(Function)	Function
invert	()	Relation
restrict	(Set)	Function
overridingUnion	(Function)	Function
curry	(Function)	Function
dispatch	(Function, Function, ...)	Function

3.3.11 RealFunction

A *RealFunction* is a function with range a subset of the real numbers. It maps all unary and binary numerical operators (e.g., log and $\times$) onto the binary RealFunction operators.

3.3.12 Array

An *Array* is a function with dom a subset of $[i : j] \times [k : l]$ for integers i, j, k, l.

3.3.13 Image

A *Image* is an array with extra properties related to its collection or creation.

3.3.14 DiHypergraph

Name	Input Types	Output Types
isPolytree	()	Boolean
isTree	()	Boolean
nodes	()	Set
arcs	()	Set
parentNodes	(Object)	Set
parentNodes	(Hyperarc)	Set
childrenNodes	(Object)	Set
childrenNodes	(Hyperarc)	Set
parentArcs	(Object)	Set
parentArcs	(Hyperarc)	Set
childrenArcs	(Object)	Set
childrenArcs	(Hyperarc)	Set
asRelation	()	Relation

3.3.15 BayesNet

Name	Input Types	Output Types
prior	(Object)	BayesArc
bel	(BayesArc)	BayesArc
asBayesArc	()	BayesArc

3.3.16 Hyperarc

Name	Input Types	Output Types
parentNodes	()	Set
childrenNodes	()	Set
asFunction	()	Function

3.3.17 BayesArc

Name	Input Types	Output Types
multiply	(BayesArc)	BayesArc
instantiate	(Tuple)	BayesArc
instantiate	(Tuple)	BayesArc
condition	(Set)	BayesArc
divide	(BayesArc)	BayesArc
marginal	(Set)	BayesArc
multAndMarginal	(BayesArc, Set)	BayesArc

3.3.18 Attribute

Attributes are elements of tuple domains. They are objects in many presentations of relation databases. They are random variables in condition probability densities and Bayesian networks. *Attributes* can have associated domain constraints, refer to measurable quantities, have units, and have physical interpretations (like row of a display screen).

3.3.19 Unit

Subspaces, frames and bases can be specialized to allow individual elements to be measurable with respect to certain known units. Distinguished (const and often static) instances of the *Unit* class include: meter, kilogram, second, coulomb, radian, degree (absolute), and monomial combinations (e.g., inch, m^2, and w/sr-Hz).

3.3.20 Dimension

Each *Unit* instance is associated with a type of measurement quantity (or dimension). Instances of the *Dimension* class include: length, mass, time, charge, plane-angle, and temperature (with many dozen more).

Section 4

User Interface

4.1 Introduction

A major objective of the Image Understanding Environment User Interface (IUEUI) is to give users flexible, simple, and powerful tools for exploring data, algorithms, and systems. Another fundamental objective is to create an interface which will be supported by ongoing and future developments in the software world at large. To achieve this, we want to capture the critical functionality of our domain in a small number of objects which are built on top of existing interface packages and interface construction toolkits. Efficiency and long term extensibility will increase by implementing features on the correct level. The same objects which will be used to implement the user interface can be specialized for users of different skill levels and objectives. This is all critical for the long term use of our environment because we can depend on dramatic changes in interface devices, voice input, video publishing, network interfaces, hypermedia databases, tools for cooperative work and communication. These are all things we want the environment to take advantage of as they are developed.

The Interface of the IUE is described in terms of three levels (Figure 4-1). The **Graphics Level** is the underlying "machine independent" package for basic display and graphic operations and telling the screen what to do. Examples would be X and Postscript. Machine independence is somewhat relative, so for now, this level can include other packages as long as they support similar functionality. The **Interface Kit Level** consists of existing packages for the creation and rapid prototyping of user interfaces and related tools on top of graphics level software. Examples are such things as Interviews, TAE, NeXTSTEP. This also must include use of tools found in the selected software development environment such as editors and debuggers. The **Image Understanding Environment User Interface (IUEUI) Level** consists of the objects in the user interface. This includes such things as object displays, plotting displays, several types of browsers, and structures for describing the interface context. The IUEUI consists of a small set of objects which can be freely combined for very powerful results. The specifications of these objects is relatively independent of the

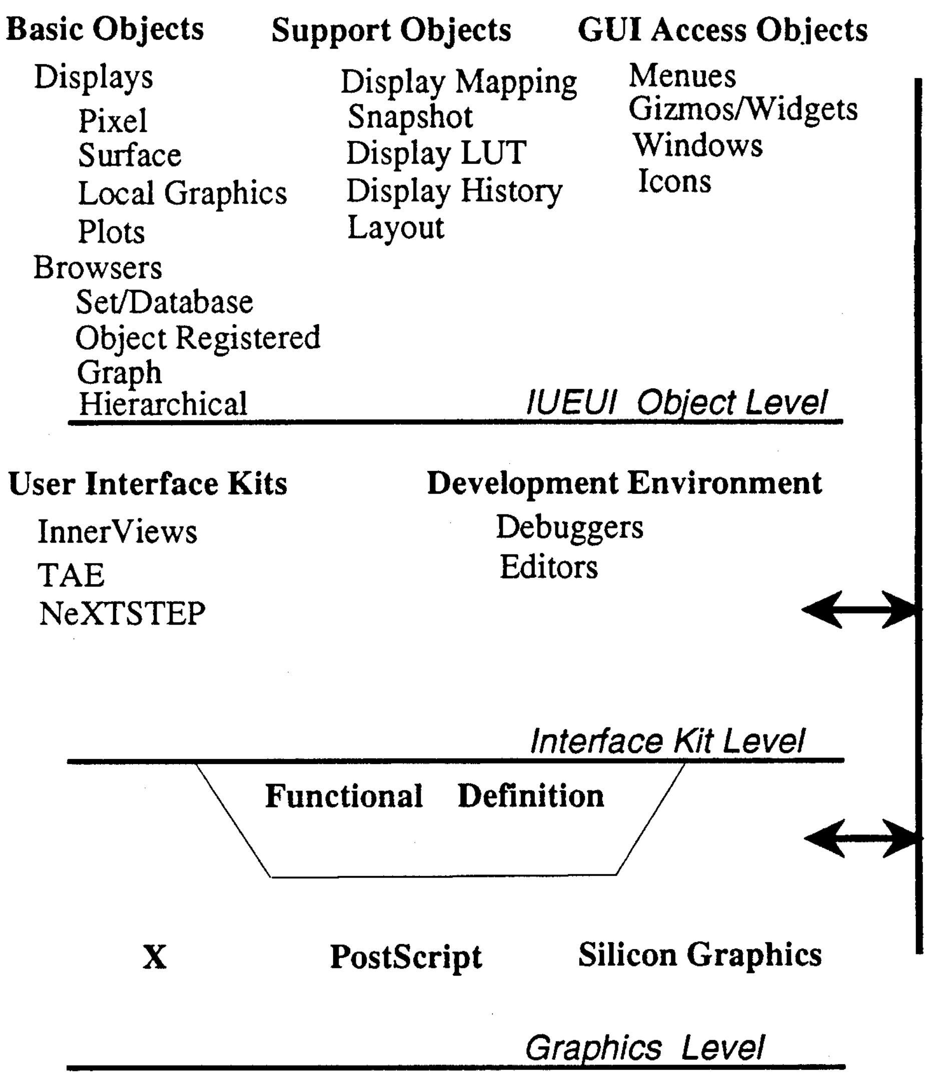


Figure 4-1: Levels of Interface Implementation.

other two levels although it will be required to show in the programmers manual how the functionality of the IUEUI objects are realized at the other two levels.

4.1.1 Paradigms

Some of the different interface paradigms we have discussed are:

- Subject/View
- Data/Flow
- Display methods as integral parts of objects

These are not disjoint in our design and each reflects critical capabilities for the IUEUI. An object-oriented methodology is essential for the interface because we don't want users to have to use literally thousands of different functions depending on the type of object they are trying to display, browse, or inspect. This is often a problem with C-based environments because users need to specify the type of value and the type of object. At the same time, we need to provide explicit, powerful tools so users can control all aspects of data visualization at anytime. We often want to display the same object in several different ways (displaying the gradient as a vector field, or mapping the different components of the gradient onto different color bands). It is not sufficient to let the user simply tell an object to display itself and then let the object "decide". It is essential that users be able to view objects in a completely controlled manner and that there are intelligent defaults reflecting common usage for less experienced users. Dataflow capabilities are necessary to allow a user to create a complex process by interactively specifying the combination of other processes and to monitor their execution.

4.2 Required Functionality

This section describes the necessary functionality for the user interface based upon the union of attributes found in several different existing IUEs and, in some cases, their deficiencies.

- **Object Display Control** We want the user to have complete control of how spatial objects (images, surfaces, features, networks, pyramids, etc.) are viewed with maximal flexibility and ease of specification. The mapping between a spatial object and a display window needs to be explicitly represented by a Display Mapping Object that can be manipulated, set, and saved. There are two basic aspects of this. One concerns how positions on the object get mapped onto positions in a display windows. For this, users want to control such things as zooming, panning, perspective, warping, and other geometric operations. For now we refer to this as the Position Mapping. The other aspect concerns how values of an object get mapped onto display window values such as intensity and color (sometimes text, icons, and graphics). For now we refer to this as the Value Mapping.

The distinction between the position and the value of a spatial object can be complicated and a user needs to be able access any attribute of an object and to display it as he wishes. For example, a discrete curve can be viewed as a mapping from integer indices onto 2D positions with respect to an image coordinate system. When I overlay the curve on top of an image, I am mapping these 2D positions along the curve onto window positions using the same position mapping that was used for the display of the image. I also would like to control the color/intensity of the display at these points based upon registered values associated with the curve (such as curvature). For example, a user might want to display an intensity image in 8-bits of green intensity and then overlay extracted curves on top of this with the display of curvature values along the curve mapped onto 8-bits of red intensity. And to say this as least as directly as the previous sentence.

- **Virtual Objects** When display transformations are applied to an object, it should not involve creating a new object: only the display in a window is generated. An example is manipulating the underlying color look up table to perform a thresholding operation. In this case, there is no thresholded image object produced, only that which is displayed in a window. This goes by many names in different systems such as Pixel Mapping Functions, Dynamic Color, Generalized Color Look-Up Tables. It includes operations such as thresholding, histogram equalization, fitting to a linear display range, overlays, and others. The display buffer is a short term memory for a view of a displayed object: we should perhaps provide routines to use this directly. Some work on the Lisp Machines would implement image processing operations by bit-blitting.

- **Commands and Intelligent Defaults to Deal with 8-bit, 24-bit, 32-bit displays**: The interface should be able to deal with different types of display devices, taking into account how deep the display buffer is.

- **Display Overlays**: It is important to be able to display extracted features and values overlaid on top of images (and other objects), as in displaying a vector field on top of an image, or in displaying extracted edges and junctions. The overlays can occur with respect to the display window (annotating an image with text) or with respect to the displayed objects (marking a displayed surface with features that occur upon it). This is one aspect of the specification of multi-object displays.

- **Linking Displays and Browsers:** Viewing transformations can be concatenated through links between displays. The view (the mapping between a spatial object and a display window) in one window can be concatenated with the view specification in another. A common example is using one window to zoom onto the display in another or using one window to display a selected portion of another. Panning and Zooming are so common they will probably be directly supported as a default or through a system level menu. It is also useful to have links between browsers, such that when an object is selected in one browser, the attributes of the selected object can be viewed in another.

- **Interface Context:** Information describing the current context of the interface state is used for intelligent defaulting. This would include things such as the current window, the current mapping, established links between windows, the thickness of lines in graphic overlays, the current displayed objects, the layout of windows and browsers on a screen, and several other things. This would be an extension to the underlying context provided by the graphics level. These contexts can be saved and read. Selection of information through browsers also has a context for intelligent defaulting. This includes such things as the most recently selected object, links between browser windows. This is especially important when interactive browsing is used for sequences of queries over a database

- **Interactive Command Language:** All display actions involving objects can be specified through an interactive command language. This should have intelligent defaults and abbreviations (such as displaying to the current window if none is specified). These commands should also be usable in code for creating scripts and general display routines. It is not necessary that all interactions take place through this command language (Some will be invoked by menus and special keys and refer to the current display context). The Interactive Command Language is especially important because it provides a functional specification of the entire interface.

- **Process Monitoring:** Monitoring the execution of a task; visualizing processing at the current locus of processing. This should be performed using the general browser class over a task database.

- **Animation Tools/Operations:** dumping window or screen output to video tape; cycling through a sequence of displays or displays written out to file; cycling through a sequence of displays; use of double buffering in the display buffer.

- **HardCopy Tools/Operations:** dumping window or screen output to paper, slides, overhead transparencies. [**Note:** should we make PostScript compatibility a requirement?]

- **Interactive Command Buffer**: The user can type-in display commands in an interactive buffer; he can cycle through commands, he can perform window-based editing operations on commands in the command buffer and then specify their re-execution. A good model is the ease of use with the Lisp Listener. We need the same functionality for interface and object interaction operations even in a C-based environment.

- **Synergy of Interface and Development Environment**: When moving from the debugger and editors for code development to the display and browsing operations of the interface, it should not feel like starting up completely different processes.

- **Object Interaction** Displays are also used for interacting with spatial objects in order to access values in them, move them around, and apply operations to them. In interactive processing, when the user clicks on the display window, the position in the window and the current object and the current object value are saved. The current object can be explicitly specified or taken from context. Disambiguation may be required if there are multiple overlapping objects. The user may be required to use a label plane (an image of pointers to objects which occupy a given position) or use geometrical data base operations in the IUE. Both are potentially expensive and don't reflect operations specific to the IUEUI but are general IUE spatial data base operations that can be accesses through the IUEUI. It is sufficient that the interface is able to return the selected object(s) and object position from the object display mapping.

- **Graphics and Text Overlays**: This involves writing and graphic drawing (both 2D and 3D) with respect to a display. This is especially useful for slide creation, documentation, and data generation. It has several modes that need to be distinguished. Sometimes we want to do this with respect to the window in which a spatial object is displayed; sometimes with respect to the displayed spatial object or object coordinate system (perspective view of an object model); sometimes we want the generated graphic display to generate and instantiate corresponding IUE objects; and sometimes we want to refer to the entire screen on which several display may be present, as in connecting features in different windows and browsers by arrows or annotations.

- **Default layouts for windows and browsers**: The desired layout of windows and browsers can be saved and be available to a user when he starts using the IUE. Several different such arrangement can be stored and brought back for different situations under user control.

- **Simplified Access to Interface Objects**: The IUE should provide simplified, interactive access to the interface objects found in GUI Kits. Such things as sliders, knobs, buttons, text input/output fields, menu creation and personalization, and icons.

- **Display History**: The sequence of display or browsing actions for a particular window or browser are saved and can be reaccessed and used for creating animations.
- **Types of Displays to be supported**: Image, Edge/Contour, Vector, 1D Plot, 2D Plots, ND Plots, Features, Surface (grid and rendered), Volumes, Color Images, Sets/Databases of feature objects [be able to refer to selected attributes], Image Sequences, Animations of Image Sets, Gray scale Images, Metrically embedded graphs (graphs showing relations between extracted image features).
- **Interactive Object Creation (Draw Objects)**: It should be possible to create object interactively. This is useful for creating sample idealized data for testing and development. Interactive Object Editing and Creation should also be supported.
- **Updating Displayed Objects**: In some cases we want a display or a browser to be updated when an object changes (such as a browser for a database of executing tasks).
- **In-code Functions for prompting user for unspecified information and other interface actions**: enable user code to refer to interface actions. Examples: A routine can specify that a particular value is to be queried from a user interactively or set from some gizmo/widget. Or that a particular browser field is to be updated when some state occurs or a process completes execution.
- **Multiple Object Displays**: A spatial object is viewed through a display window by a mapping from the spatial object onto the display window. It is important to be able to map several spatial objects onto the same window at the same time or through the incremental creation of a display. Examples are for such things as mapping different images in a stereo pair onto different color planes; overlaying extracted features onto an image (or arbitrary surface). For operations such as overlays, transparency, Color, Cycle, flashing, display buffer animation.
- **Mensuration tools**: rulers, grid overlays, flashing them; orientations of rulers under zoom-links between windows; and cursor type and the use of multiple cursors. Whether the measurements are being made only relative to the 2D display window or with respect to the spatial object. These can be built on top of the basic interface capabilities and the display of IUE objects (in particular, the interactively functionality of the display object and IUE objects such as bit-mapped regions, line-objects).

- **Incorporating Hardware Accelerators**: The interface (and the IUE in general) should be able to deal with different hardware accelerators and a distributed computing network.

- **Interactive programming tools**: Several questions remain to be answered: Can these be as in Khoros (an image processing system with data flow programming environment freely distributed by University of New Mexico)? Can these be developed from the graph browser and icons? How are DataFlow transformations handled?

- **Device Menu Shortcuts (Bucky Windows)**: Many menu operations should be mapped to combinations of key and mouse button processes. The user should be able to modify and extend the default mapping.

- **Access to and Integrated use of Established Visualization Packages**: There now exist several data visualization products, (Plotting package in Mathematica, packages from Precision Visuals). We should select those which are relevant and supply interfaces to them. In fact, we will find ourselves recreating this functionality (especially for plotting). There are, however, problems with data type compatibility, speed, selection tools to get exactly what we want displayed from these packages in the IUE. Much of the interactivity may not be obtainable and the feel may be considerably different than the IUEUI.

4.3 IUE Interface Objects

We break the IUE interface objects into three basic classes (See Figure 4-2). The first class consists of displays and browsers. These are the basic tools for viewing an object and inspecting its symbolic attributes and relations (There are many commonalities between these objects that suggest a meaningful and general IUE interface object). The major portion of what a user does with the interface will be based upon these objects. The second class are the objects commonly used in the supporting user interface mechanisms provided by the graphics and toolkit level (menus, widgets, icons, etc.) but with simplified commands so they can be manipulated directly by IUE users. The third class are support objects for such things as describing the current interface context, the mappings from an spatial object onto a display window, links between IUE interface objects, animation files, and several other things. Many of these are not necessarily objects, but common data structures.

4.3.1 Object Display and Browsers

- **Object Displays**: (See Figure 4-3) This is for viewing objects which have coordinate systems associated with them and mapping them onto a 2D display.

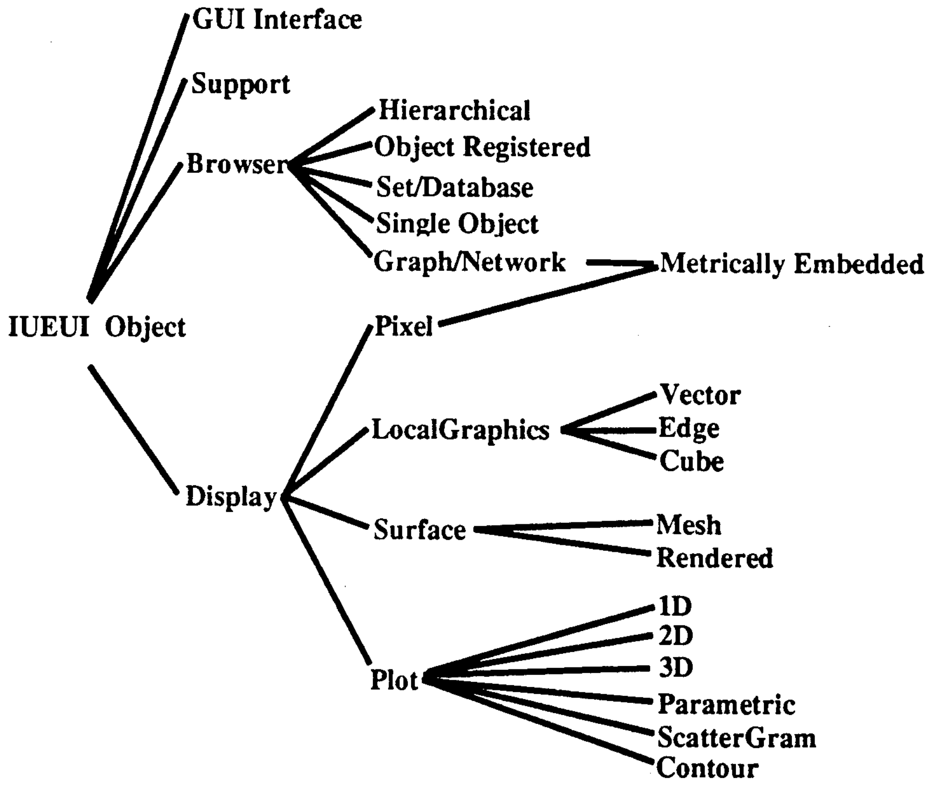


Figure 4-2: IUEUI Object Hierarchy.

It includes such things as images, curves, regions, object models, surfaces, vector fields, etc. They support several types of operations for controlling the mapping of an object to be viewed in a window and for interacting with a displayed object. There are several subclasses of displays that will appear to the user to occur in the same type of window. They are primarily distinguished by the types of methods they understand and all inherit a large number of similar methods from the general display class. For example, the overlay method means something different in the context of a surface display than in the context of an image display. The **pixel** display class is for viewing images and image registered features. The **local graphics** display class displays objects by mapping their values onto graphic objects such as lines and cubes. Examples are displaying vector fields and edges. The **surface** display class is for displaying objects that get mapped onto mesh or rendered surfaces. There are several different types of **plot display**: 1D, 2D, 3D graphs, histograms, scatter grams, perspective views of functions and tables. **Note:** We may implement plotting windows from existing packages (e.g., Mathematica). It will be complicated to extend or integrate such plotting/visualization packages with the other capabilities which will definitely be associated with display-windows (such as object interactivity and selection, function application, window-linking). We may find that we can live with this or we will need to implement some subset of these plotting packages directly in our interface.

- **Browsers**: These are used for actions such as queries over set of objects, determining and inspecting relationships between objects, process monitoring, and inspecting values in an object. There are 2 different types of browsers: **Field-Browsers** and **Graph-Browsers.**

  Field Browsers consist of a regular array of fields. Fields can be filled with text, icons, colors, colored text, text in particular fonts. Fields can have actions associated with them when they are selected or a user changes the values in them. We distinguish between four types of Field browsers which inherit from the general Field browser class:

  - **Set/Database Browser**: This is presented as an array of fields. Each row of fields corresponds to selected attributes of a particular object and each column corresponds to common attributes over the set (or database) of objects. An example would be browsing the database which describes the current active object in the IUE to find the most recently created image from some operations (See Figure 4-6).
  - **Single Object Browser**: Each row corresponds to the value of an attribute for an object. This is used for inspecting a single object (See Figure 4-7).
  - **Hierarchical Browser**: Useful for text based inspection of graph structures and trees. When an item is selected, the related items (along some relational dimension) are displayed in the next column (See Figure 4-8). [**Note**: A good example is the directory browser on the NeXT machine].
  - **Object-Registered Browser**: This contains values extracted from a spatial object, such as the intensity values in some square neighborhood of an image. Depending on the dimensionality of the object (or relationships between component objects), this can be presented as a 1D array, a 2D Array, or multiple 2D arrays and describes curves, images, image sequences, pyramids. There are restrictions on whether it is possible to interactively change values in the fields of an array browser. It should be possible to apply operations directly to the values in the array browser to see the effect of an operation in a restricted neighborhood of an object (See Figure 4-5).

- **Graph Browsers**: These are for the display of graphs and networks, generally representing an object as a node and links to describe relations to other objects. Nodes are similar to fields in field browsers and can be filled with text, icons, colors, colored text, text in particular fonts. Nodes can also have actions associated with them when they are selected or a user changes the values in them. Links can also be colored and selected. A typical use would be for the display of a constraint network (See Figure 4-9). **Note:** For complicated relationships or large sets of objects, these can become very complicated and we may need a way (e.g., GRASPER) for segmenting nodes and links into spaces].

An important type of graph and graph browser is a **metrically embedded graph** wherein the nodes (and perhaps links) are restricted to occur at positions with respect to a coordinate system. This type of graph inherits properties from both the Graph Browser and a general spatial object which can be viewed in a display window. An example would be an image registered network which describes potential links between extracted features for displaying grouping operations. An important attribute of metrically embedded graphs is that they can be viewed as an object display for operations such as zooming and having access to the underlying context in an image. [**Note**: We may also want to distinguish tree graph browsers.]

4.3.2 Simplified access to GUI objects

- **Gizmos and Widgets**: The IUE should provide simplified, interactive access to the interface objects found in GUI Kits. Such things as sliders, knobs, buttons, text input/output fields, menu creation and personalization. This will involve commands for creating gizmos and widgets, for positioning and scaling them, for attaching them to parameters, for reading and writing to them. An example would be creating a slider and then getting values for an interactive thresholding operations from it.
- **Menus** The IUE should provide simplified interactive access to menus in the GUI kits. This involves being able to extend menus, create pop-up menus, associate actions with menu items. A critical design task is deciding what goes into system level menus and how they are organized
- **Icons** The IUE should provide simplified interactive access to icons in the GUI kits. These may not be flexible enough for our needs [**Note**: Icons in Khoros representing processes can be connected together to form a graph for algorithm creation and process monitoring. We may be able to obtain the same functionality by using a graph browser with icons at nodes].

4.3.3 Support Objects

There are also several objects that are used and manipulated as part of the interface that we will refer to:

- **Display-Look-Up-Table**: A generalization of a color look up table that describes how to map object values onto screen values. It can also include functions.

- **Object-Display-Mapping**: A structure which describes the mapping from an object onto a display. This includes both the position and values of how the object is displayed and a reference to a particular Display Look Up Table.

- **Object-Browsing-Mapping**: A structure which describes the mapping from an object or database onto a browser.

- **Object Display Links**: A structure which describes the concatenation of a display or browsing operation between IUE interface objects. Thus a link between display windows w1 and w2 with an associated zoom and pan would display an object in w1 with w1's object display mapping and then display the same object in w2 by concatenating onto the object display mapping for w1, the specified zoom and pan operation.

- **Interface layout**: A structure which describes the object instances in a particular instantiation of the interface. Users may prefer different interfaces (arrangement and instantiation of the basic IUEUI objects) depending on the task or level of sophistication.

- **Display Context**: A structure which describes current context for a display. Such things as the current window, the current object, the current object display mapping, the current display command, the current mouse-selected object position and value, and others. Display operations can use defaults based upon these.

- **Browse Context**: A similar structure for browsing operations. Such things as the current browser, the current data base, the query history, and others.

- **Display Snapshot**: What is produced when the current display is written to a file. It is just what appears on the screen and not the actual objects.

- **Animation File**: A sequence of display snapshots.

4.3.4 Interface to Arbitrary Interface Devices

In the development here, we are assuming a very limited set of interface object: a mouse and a keyboard. By the time the environment is commonly adopted, there will be a much wider array of objects such as: voice input, gestural recognition (data gloves), and perhaps virtual reality displays. What tools will we provide to interface such devices with the IUE? At this point, we are raising this as an important design issue. A rough cut was made at this earlier by suggesting objects

- Keyboard object

- Pointer object

- Mouse object
- Trackball object

with methods such as Open-Device, Get-History, Close-Device, Attach, Get-Status, Handle-Event. More extensive work will be needed in this area, perhaps requiring special IUE interface buffers and IUE access to Graphics Level event handling routines.

4.3.5 Total Synergy between IUE object methods and the IUE Interface Operations

Any of the operations that can be applied to objects as part of the IUE should also be applicable to object prior to displaying or browsing them. In the examples below, because the syntax for this is not specified, there will be some confusion. For example, how do I refer to the registered curvature values associated with a curve object if I want to display them? What is the query I use to find all curves greater than some length prior to displaying them? Using the interface requires use of objects and operations in the IUE, notably database operations and queries; referring to a particular set of attributes of an object; selecting a portion of an object; applying a transformation to object values; and operations for combining objects. Much of this comes for free in LISP. It may require building a special parser for non-interactive programming environments with an extensive library of functions. (**Note**: All IUE objects will also require browse and display methods) .

4.4 Object Display

The object display is for mapping an IUE object onto a 2 dimensional view with methods for controlling how this is done. It is a major object and is specialized into some different types of displays which have different inherited methods from the general display object. A display is also recursive in that it can consist of several displays. Performing and interacting with an object display takes place in an object display window. From the user point of view, an object display can be thought of as a window with different component displays in it.

4.4.1 Appearance

The layout for the object display window is shown in (Figure 4-3). It is a rectangular window with

- a title bar (which can be colored or patterned)

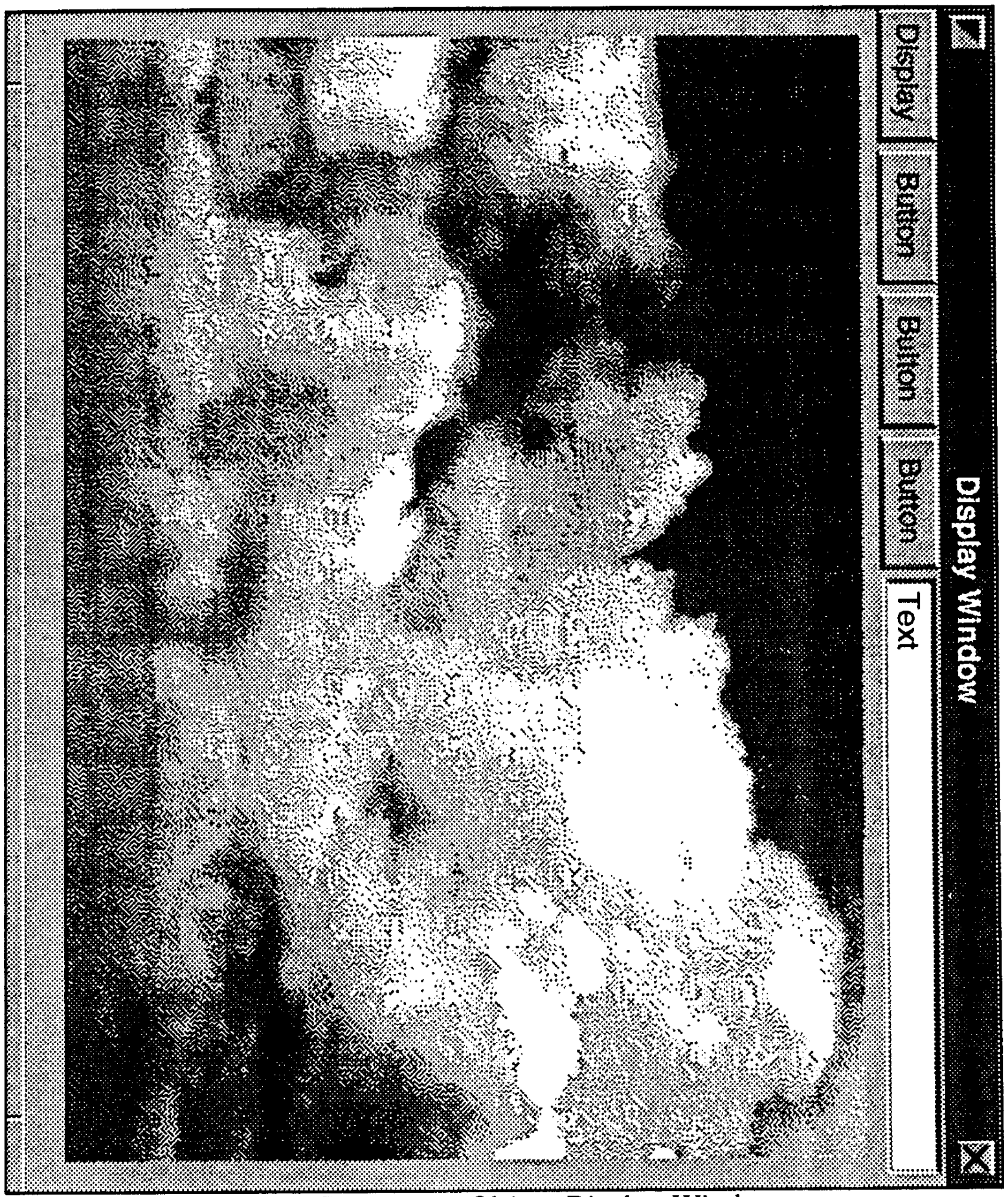


Figure 4-3: Object Display Window.

- a close box for closing the window
- an iconization-box for turning the window into an icon and saving it's state
- horizontal and vertical scroll bars for panning
- a resize box for changing the size of the window
- a status bar containing information about the current display object(s) and the last selected object location and value
- a button bar for actions such as display of the current browse selected object.

A window can be selected by clicking anywhere on it with a mouse. It then becomes bound to the variable named *current-window* for future displays and interactions. Thereafter, when the user clicks in the display portion of the window, he executes the interactive methods which defaults to displaying the current object location and value (the user can associate arbitrary interactive methods with mouse actions). The current window has a distinctive highlighting of the title bar as do any window to which is is linked. As an option, the user can hide other windows except for the current window and windows to which it is linked. The window can be repositioned by clicking and dragging on the title bar.

There are two other associated windows for interacting with the display in the current window:

- **Interactive Command Buffer.** This appears like a WYSIWYG text editor. Textual outputs can also be written to the Interactive Command Buffer. It has a vertical scroll bar for accessing previously written commands and allows operations like cutting, pasting, etc. It is something like a limited Lisp Listener for interacting with objects and displaying them. A nice feature it should incorporate is the use of shift-return in Mathematica: a command is only executed when the user hits shift-return. This makes it possible to create complex multi-line operations and also breaks the display command history into chunks of related actions that a user may want to access for later editing. (**Note**: It may be a Lisp Listener or something very similar if the development takes place in an interactive programming environment.)
- **Display Tool Box.** There are many familiar interactive controls for displays and visualization, such as interactively manipulating the object-value to screen-intensity function by interactively shaping the function; selecting color look-up tables; modifying color look-up tables; interactively building display commands using templates or command browsers; floating tool palette of interactive drawing tools; etc. The Display Tool Box is a menu of such tools, organized into functional groupings of interactive tools for manipulating the current display.

From a functional point of view, it allows potentially redundant access to the display methods without using the interactive command buffer. It is somewhat like the system control menu on the Macintosh and the system preference menu on the NeXT machine. In the Figure 4-4, the large buttons on the right are some of the different modes of interaction. The interaction field on the left has the corresponding interaction tools for a selected mode. The ones shown here are existing ones on the NeXT for manipulating the color look-up table. The order of the buttons in the tool box should be settable by the user. Users can also select particular interaction tools and have them occur as floating palette (in cases where the user want to interact with multiple tools from different sets of tools at the same time).

(**Note**: We probably need a similar menu for setting up system defaults and initializing characteristics of the IUE: initial layout, font selected, level of expertise, etc.)

Some of the tools sets that should be included in the tool box are:

- Interactive Selection and Modification of the the current color lookup table and display mapping function; cycling through different color look up tables
- Interactive Selection and modification of a display's attributes using browsers for the display window's attributes that can be used for changing attributes of the window and display; Browsers over the windows that the window has links with
- Interactive Display Command Creation such as Browsers for selecting existing position and value functions to be placed into commands; available options for the different display commands and their current defaults presented in template form
- User-created gizmos/widgets: sliders, buttons, knobs that the user has created
- gizmo/widget arrangements of which a user is particularly fond
- Interactive gizmo/widget creation using gizmo selection-wells
- sending displays to a printer
- animation creation and playback

Note: We can either associate the command buffer and tool box with each display window or else have them automatically linked to the current window. The second alternative is simpler, though it may have some context problems since displays will always default to the current window. It may be best to default to one command buffer and one tool box, but to have their context shifted in terms of setting to different selected tools when different windows are selected.

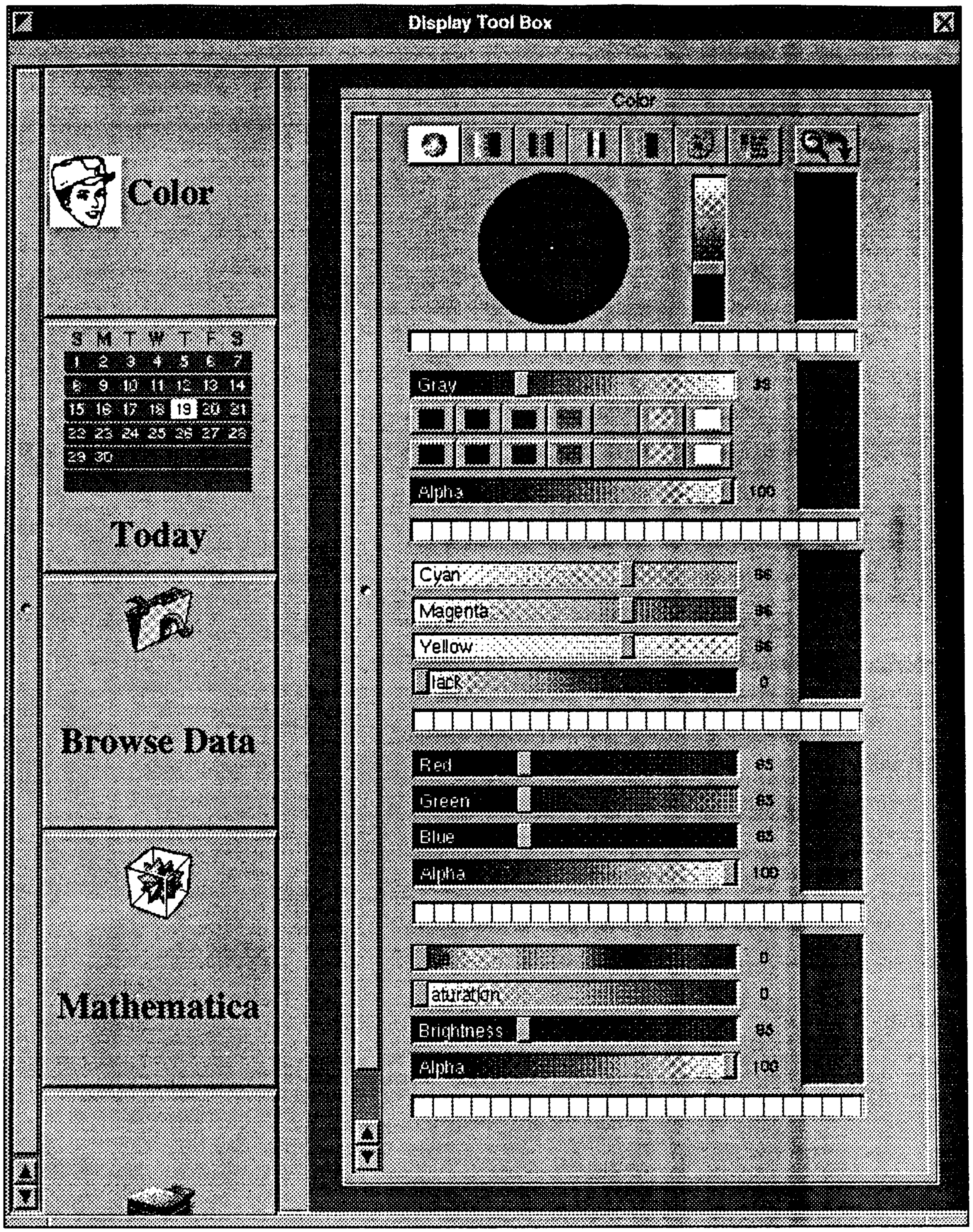


Figure 4-4: Display Tool Box

4.4.2 Attributes and Methods

Class-Name Object Display

Description An *Object Display*

Sources/existing implementations

Superior(s) IUE Interface Object

Component Class(s)

Associated Class(s)

Slots

Screen-Size: Size of window in screen coordinates. Returns a 2D integer vector (maybe a specialized struct)

Screen-Position: Position of upper, left hand position of a display window in screen coordinates. A 2D integer vector (maybe a specialized struct)

Internal identifier: unique number associated with each display window

Name: Name of window: a string

Title-Bar-Color: Color of the title bar

Title-Bar-Font: Font Object

Cursor: Any of the set [cross-bar, cross-hairs, arrow, user-defined (**Note**: These should be definable as IUE Objects: lines with constraints between them mapped onto the overlay planes or bit-mask regions).

Children-Windows: A list of display-link objects describing displays which are linked to this display

Parent-Window: The display to which the display is linked

Current Object Display Mapping: An object view mapping object describing the current mapping of an object onto the display. This includes pixel scaling factors

Current Display Command: A set of strings for the current display command

Object-location-list: A list of locations selected by clicking the mouse on the display: it inverts the specified object view mapping (the default is the current one) to determine the location in the specified objects

Object-value-list: A list of values selected from an object using the object-location-list

Window-location-list: A list of locations selected by clicking the mouse on the display. The locations are in screen coordinates

Display History: A Database

Current Display Mapping Table: A display Mapping Table structure

Methods There are several methods for displays and they are organized into different groups (A good way to see the effects of these methods is to look at examples from the section on the command language):

Methods for Manipulating the Current Object display position mapping: This includes operations such as panning, zooming, perspective views, and warping. These are methods that control how positions in the specified object(s) get mapped onto a display window.

Methods for Manipulating the Current Object value mapping: These include operations such as overlays, mapping onto different color bands, transparency, and others. These are methods that control how values in the specified object(s) get mapped onto screen attributes such as color and intensity.

Methods for setting the current display mapping table: These include how to configure planes in the screen buffer for the display of color images; how many panes to use for overlays; particular functions and conditions to apply to object values prior to display.

Methods for Screen Attributes: These involve controlling attributes of the window the display is mapped onto and includes such things as position, size, attributes of the title bar, event handling for the mouse.

Methods for Links: Linking display transformations in different windows. Operations include creating links and associating position and value mappings with the links.

Methods for Interaction: These involve interaction and manipulation of displayed object(s) in the display. Operations include recovering object position and value from a mouse click, applying functions to selected objects, applying functions using selected information.

Methods for History: Methods to coordinate displays overtime, such as cycling through an image sequence, playing an animation of displays.

Methods for Graphics: These involve accessing display registered graphics packages for drawing lines, text, and other things. These occurs in four different modes: 1) relative to the window of the display; 2) Relative to the entire screen; 3) the specified object or coordinate system; or 4) for instantiating IUE objects corresponding to the graphic displays.

Methods for printing and writing to file, animation: It should be possible to send the view of any object to a hardcopy print device or to files for later redisplay or printing.

Manipulating the Current Object Display Position Mapping

zoom (window-zoom)

Arglist x-zoom-factor y-zoom-factor

Return Type None

Description Specifies the scale of mapping from an object onto a window

Exceptions

pan (window-pan)

Arglist x-translation-factor y-translation-factor

Return Type None

Description Specifies the moving the position of a displayed object in a window

Exceptions

Matrix

Arglist Homogeneous Viewing Matrix

Return Type

Description

Exceptions

Position-function

Arglist Code Chunk

Return Type None

Description A chunk of code which computes a position from positions or values taken from the specified display objects. These are referred to as dummy variable by placing ".value" or ".position" after the object name (**Note**: what should be done if the values and positions are not just scalars?). As the display processing iterates over the specified object, the code chunk is applied to the specified values and positions prior to the display to generate a new position. In Lisp, these can be expressed as functional closures and are pretty easy to manipulate. In C, we may have to provide a compiled library of code chunks that can be used in this way or with a parser.

Exceptions

Warp

Arglist

Return Type

Description

Exceptions

Manipulating the Current Object Display Value Mapping

linear-mapping (linear-map)

Arglist object-value-min object-value-max screen-value-min screen-value-max

Return Type none

Description Will map the specified range of object values linearly onto screen intensity values

Exceptions screen-value-min and screen-value-max will have default values

values

Arglist a method for extracting values from a spatial object

Return Type none

Description

Exceptions

value-function

Arglist A chunk of code which computes a value from values taken from the specified display objects. These are referred to as dummy variable by placing ".value" after the object name. Essentially, as the display processing iterates over the specified object, the code chunk is applied to the specified values prior to the display. In Lisp, these can be expressed as functional closures and are pretty easy to manipulate. In C, we may have to provide a compiled library of code chunks that can be used in this way or with a parser.

Return Type none

Description see examples throughout this document

Exceptions

overlay-object

Arglist an object (this could be a virtual object)

Return Type

Description

Exceptions

Methods to set the Display Mapping Table

rgb-24, rgb-16, rgb-8

Arglist none

Return Type none

Description Specifies that the display of colors is mapped onto red green blue values on the display

Exceptions There are going to be several variants of this depending on how we want to specify allowable overlays values, deal with transparency. We should copy something simple from graphics. The different components can then be referred to as :red, :green, :blue.

overlay-color

Arglist Any from the enumerated set (red,green,blue,clear,....)

Return Type none

Description will perform the specified display in the overlay plane as a solid color.

Exceptions Assumes the display value is binary; if not, 0 gets mapped onto no display and other values get mapped onto display in the overlay color. The different overlay colors can be referred to by name or an associated number.

yiq

Arglist

Return Type

Description

Exceptions

hsu

Arglist

Return Type

Description

Exceptions

CLUT

Arglist

Return Type

Description

Exceptions

Transparency

Arglist

Return Type

Description

Exceptions

Flash

Arglist

Return Type

Description

Exceptions

Methods for Links between displays

create-link (link)

Arglist Interface-object1 Interface-object2 display-mapping-object

Return Type

Description

Exceptions

set-link

Arglist Interface-object1 Interface-object2 display-mapping-object

Return Type

Description Sets (changes) the display-mapping-object associated with the link between interface-objects

Exceptions

un-link

Arglist

Return Type

Description

Exceptions

Methods for Interaction

initial , always, continuous

Arglist Interface Command

Return Type

Description

Exceptions

interactive-function-selection

Arglist a set of Interface commands indexed by a number

Return Type

Description For an interactive function, programs are associated with different numbers. When a key is selected, the corresponding function is applied to values in the different lists of values obtained by mouse clicking.

Exceptions

clear

Arglist none

Return Type

Description Clears the position and value lists (probably need other for removing some number of items

Exceptions

4.4.3 Local Graphic Displays (vector, edges)

Local Graphic Displays are a subclass of object display, but instead of mapping an object attribute onto a screen intensity or color, it will display a parameterized graphic, such as a line, a square, a perspective view of a cube, or something from a library of such displays or a user specified one. A common example is a vector display which will map each component from a pair of images onto the x and y components of a vector, usually displayed as an overlay on top of an image. Other types of visualizations are possible. For visualizing three dimensional attributes in register across an image, the user can display little unit cubes with their orientation computed from the specified components of display. The graphic display can be a piece of graphics code which will be positioned to the projected location of the pixel. For example:

```
[lg x-component y-component :graphic-type xy-vector :overcolor-plane red]
```

or the command language could include equivalent expressions such as

```
[v x-component y-component :overcolor-plane red]
```

will take two images and use them to specify the respective (x,y) components of vectors displayed in the red overlay plane at each point.

```
[lg x-component y-component
        :graphic-type xy-vector
        :x-increment 10
        :y-increment 10
        :value-function [+ x-component.value y-component.value]
        :linear-mapping 0 20 *min* *max*]
```

This displays a vector at every 10th pixel with the intensity of the displayed vector determined by summing the x and y components and then linearly mapping these between 0 and 20. Note the use of methods from the general display class.

```
[lg x-component y-component gradient-magnitude
        :graphic-type xy-vector
        :x-increment 10
        :y-increment 10
        :scale .5
        :value-function gradient-magnitude.value]
```

This displays the same vector field, but uses the third component to determine the intensity of the displayed vector. The scale method is expressed in object coordinates (accounting for the size of an object pixel relative to a screen pixel). There would also be methods for displaying vectors specified in (r,theta). (**Note**: we could have a method xy-vector-normalized).

Edges are similar to vectors in drawing lines corresponding to the edges. Different types of edge displays should be distinguished:

- Mapping onto horizontal and vertical edges in the cracks between pixels
- Placing a single edge at the center of a pixel with it's orientation determined by the specified components objects

One way for displaying 3 dimensional surface orientation from images which contain the orientation angles:

```
[p image]
[lg angle1-component angle2-component
        :graphic-type unit-cube-perspective
        :overlay-color red
        :x-increment 10
        :y-increment 10]
```

4.4.4 Surfaces

SRI for terrain views of surfaces

Modification of existing package

4.4.5 Coordinate Transform Network of Images

There are several objects and displays which involve sets of related images, such as stereo pairs, motion sequences, pyramids, and mosaics. In some cases the relationship between the images is expressed by an explicit coordinate transform between the components of the object.

These displays can be handled in several ways:

- display each image in a different window and have them associated by links
- display each image in the same window but scale it to be appropriately registered with a selected scale and cycle through the display history
- display each image in the same window but choose a single scale for all of them and use general display mechanisms for indexing through all the displays.

4.5 Plotting Displays

Much of the functionality for plotting displays is defined in existing packages such as Mathematica. Mathematica has different types of plotting displays with "methods" corresponding to keyword based argument passing. It would be nice to be able to use something like this.

There are some particular things that need to be addressed in using such a package that may require recoding into our environment:

- It should be compatible with the general display methods for such things as interacting with the plot display using a mouse; for the use of different colors; for being able to display.
- It should be fast and not require time consuming conversion between the data formats of the plotting package and IUE objects.

Different types of displays include 1 dimensional plots, 2 dimensional plots, 3 dimensional renderings, scatter grams, density plots, parametric plots and contour plots.

4.6 Field Browsers

Browsers are generally used for interacting with the textual, symbolic and relational aspects of objects. Different browsers are used for different types of objects. There is one type for browsing a set or database of objects (set/database browser). There is another for inspecting attributes of a single object. There are two different types for viewing networks and graphs (graph browser and hierarchical browser). There is another type for inspecting spatial object which have coordinate systems or relations between coordinate systems (Object registered browser).

(**Note**: Should browse methods be specialized based upon the type of object being browsed? Or how objects get mapped onto a particular browser? One type of interaction for dealing with lists, a structure, etc. How do we conform to the object inspectors in the development environment?)

Browsers have many similarities with displays. They can be linked. They have something like a value-function or display mapping object in that the characteristics of the browsers field (such as the text size, font, background color, display of an icon) are determined by the object being browsed.

(**Note**: How should links between display windows and browsers be handled?. When a display is updated: the browser can be updated: when the browser is changed a value in the browser is changed, a display can be updated.)

Browsers are built from several component objects:

- A field appears as a rectangular box which has slots for
  - o a background color
  - o a text string
  - o a text-size, text-font, text-color
  - o a type of boundary
  - o a screen-size and screen-position
  - o an icon
  - o a field mapping object which describes how to map attributes of an object to the different of the field. For example, in the object registered browser, a field could contain a number colored as the intensity of the corresponding location in an image. Or the background of the field could be colored at this intensity and the text string shown in another color.

  A field also has a field-selection-action that is performed when the field is selected by a mouse-click. An action could be to bring up a browser on the object in the field that was selected. When a field is selected, what happens is much like when a mouse clicks a pixel in a display in the interactive mode. The location and value are stored in a list for later use.

- Fields can be organized into connected **horizontal or vertical field groups** where in each field as a unique index in the Field Group. The fields in field groups will generally have different object mapping functions. An example comes from the object registered browser in which a given field group can correspond to registered values from different objects. For better visualization, these could be displayed in different colors, fonts, etc, in addition to their position in the field group. A field group can also have a distinct boundary.

- Field Groups can be organized into field matrices where in each group as a unique index set in the field matrix. This is used for mapping objects onto the matrix. Unless otherwise specified, the component fields of a Field Matrix will inherit the field attributes of the Field Matrix. There are constraints on the allowable sizes and positions of component fields.

- Field Matrices can be scrollable as a way to control the mapping of an object (or object set) onto the Field Matrix.

4.7 Object Registered Browser

The object registered browser is used to inspect the values in a neighborhood of a spatial object. A common example is inspecting the image values about a selected point. It is very much like the display of a spatial object in a display window, but instead of the values being mapped onto window positions and screen intensities and colors, things are usually mapped onto field locations and text. It is also possible to map onto general field values such as colored text, changes in font, colors, and icons.

The layout of an object registered browser as it would appear for an image sequence is shown in (Figure 4-5). The fields display the specified values in a particular frame centered on the specified frame number and (x,y) position. In the actual display, the field which corresponds to the location of the array browser is shaded or highlighted. Spatial Directions are appropriately registered: up in the image means up in the array browser. The Object to Field Mapping Object specifies for a particular object what type of field mapping is used for it's specified attributes or values. This can be a user specified function or from a library of existing functions. Users can select the type of font, text size, color, icons, that they want values in an object mapped onto.

For using the browsing an image and a segmentation:

```
[browse-object-registered
       image label-plane
       :field-action [browse *b3* [get-value label-plane
                                             object-location.x
                                             object-location.y]]
```

This indicates the importance of being able to access the object values and locations from the selected browser fields and the use of a object browser mapping object. This is very much like what occurs with mouse clicking in a display window for a displayed object: the values go into the object-value-list and the corresponding object locations into an object-location-list.

(**Note**: how would we perform array browsing on a pyramid? What is the local neighborhood? At given point, display the values in the children level. Assume there is an explicit coordinate transform between the different levels. The dimension button could then correspond to the things at different levels. What if the relation is a sequential one over time local neighborhood structure? Objects are related by some coordinate transform between the levels.)

(**Note**: how would we use a region to define the shape of the neighborhood? Use an object to perform a composition operation with another object? Use a curve to pull values out of an image? Use a containing volume, minimum bounding rectangle, minimum bounding cube and do regular array browse with respect to that?)

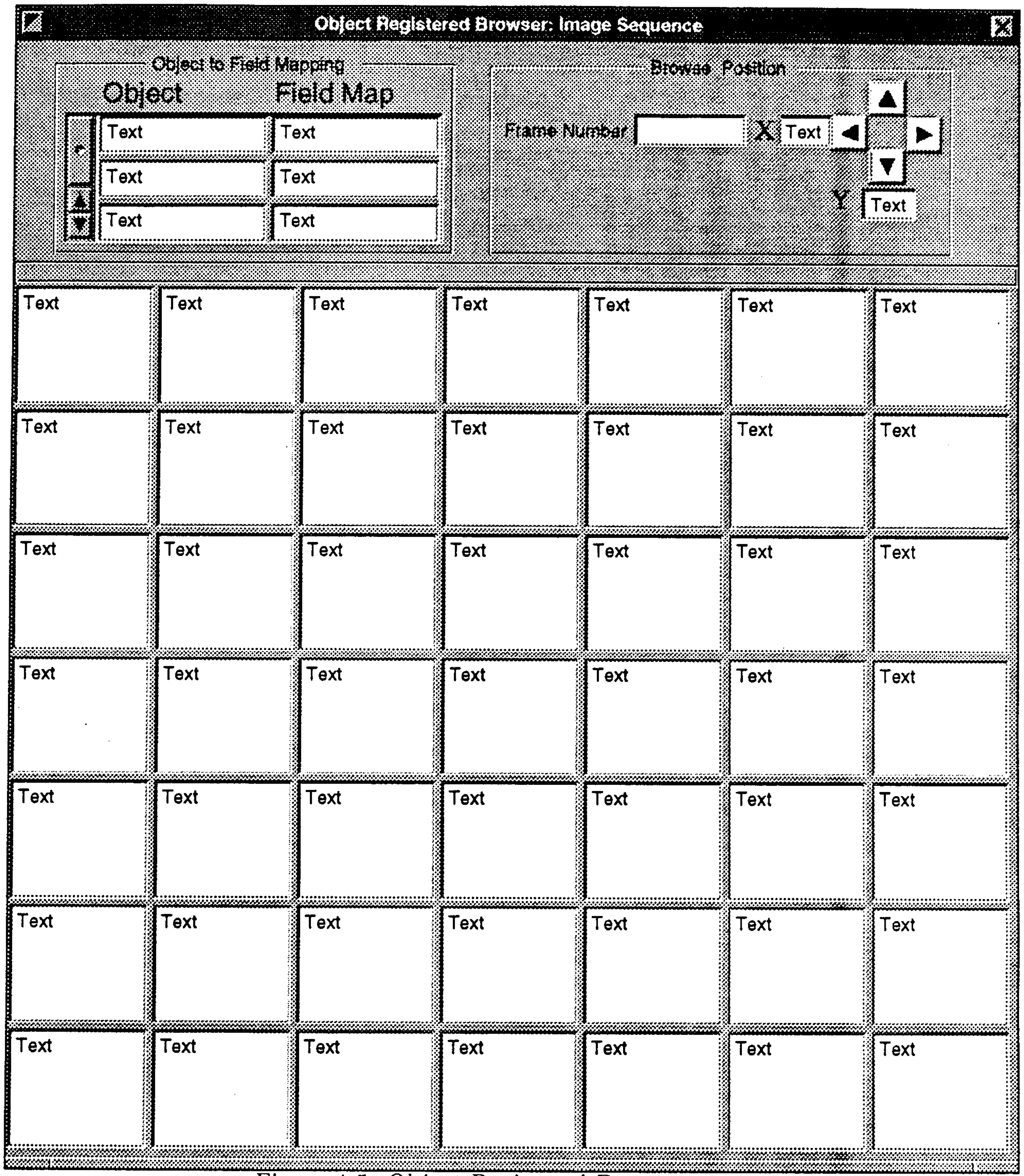


Figure 4-5: Object Registered Browser.

4.8 Database/Set Browser

An example Database/Set Browser is shown in Figure 4-6. It consists of three basic parts. A field of button for interactively building queries, selecting objects, and applying operations. A text input field for typing in pieces of text to be used in queries. A field matrix in which rows will correspond to an object and columns to attributes the objects have in common.

For example, suppose I wanted to find a result that I generated but couldn't recall. In general, it would be a good idea to keep a browser over the set of active objects (let's call this the environmental Data Base – EDB). I would set this up with the command:

```
[create-db-browser EDB :attribute-fields name time-of-creation from-function]
```

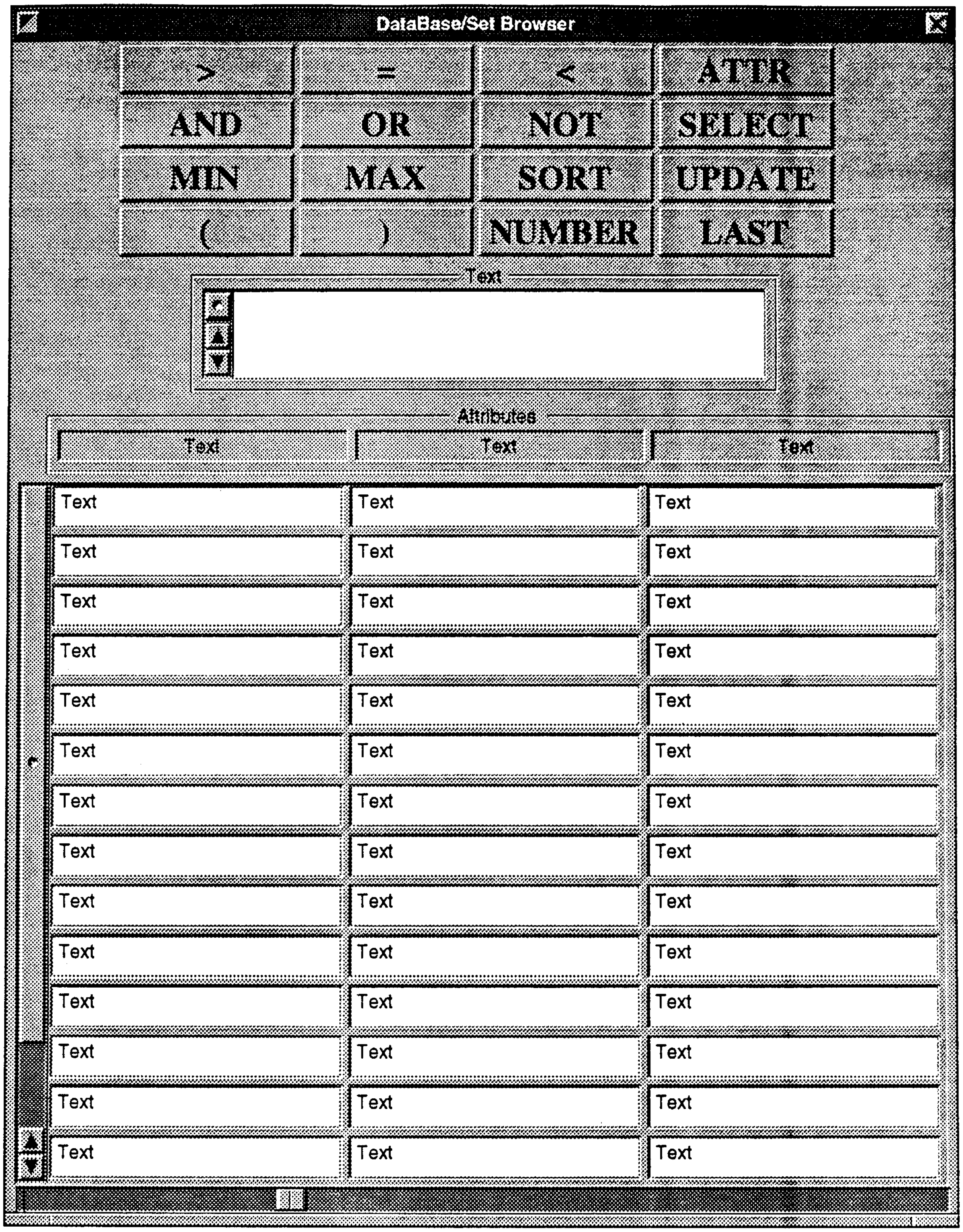
Sometime into my work, I want to find a flow-field that was the result of an amazing-function. I would type in the text area "amazing-function" **Note**: for text-matching, substring matching could be used], click it, click =, click the column attribute field labeled "from-function", click **UPDATE**. Then the browser would list all the objects which had been created from the function *amazing function.* To find the most recent, I would click the attribute-field labeled "time-of-creation" and then click **GREATER-THAN**, **SORT**, **UPDATE**, and the topmost field would contain the most recently created object from this routine. This is also bound to the variable *current-object* and would be displayed as a default.

(**Note**: Perhaps, as a query is being created interactively, it should be displayed in an interactive command buffer. If so, how should this related to the text input field in the browser?)

For each query, a set of selected objects is found. This set can be used for further queries. Sometime people make mistakes or want to return to a previously determined set. The **LAST** button is for this. We could save the current sequence of sets and continually back up through them by repeatedly clicking the **LAST** button (**Note**: This may be too costly).

It is also possible to associate with the text field selection action, the operation to browse the selected object in a separate object browser.

```
[browse EDB :selection-action [browse *b1* *selected-item*]]
```



Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text

Figure 4-6: Set/Database Browser.

4.9 Object Browser

The object browser is a simplified version of the Database/Set Browser with out mechanisms for interactive queries. It is used for inspecting the attributes of a single object. One column is used for the names of attributes and the other column is used for their values.

4.10 Hierarchical Browser

The hierarchical browser is used for the inspection of graphical and network objects. Each column corresponds to a set of objects. When an object is selected, the types of relations (arcs) associated with the object are displayed in the *Current Arc Browser*. For a selected type of relation (arc), the related objects are then displayed in the next right column. This can then be repeated any number of times. It may be useful to have a field selection method to browse the currently selected object.

4.11 Graph Browsers

We need to define how to display IUE constraint networks. This includes: being able to zoom these in something like a display window, coloring the nodes based upon their attributes, and mapping them through a pixel mapping function.

Graph browsing a relational object or a graph will be set up using operations such as, add-node, link-nodes, remove-nodes, add-arc, remove-arc, and interactive creation.

4.12 GUI Object Access

We need a general create-gizmo command.

It would be nice if the SRI model for menu definition for the environment could be set up in a browser and modified interactively.

4.13 Associated Objects

```
link-transformation object
parent display
child display
Object
Object-View mapping to be concatenated
```

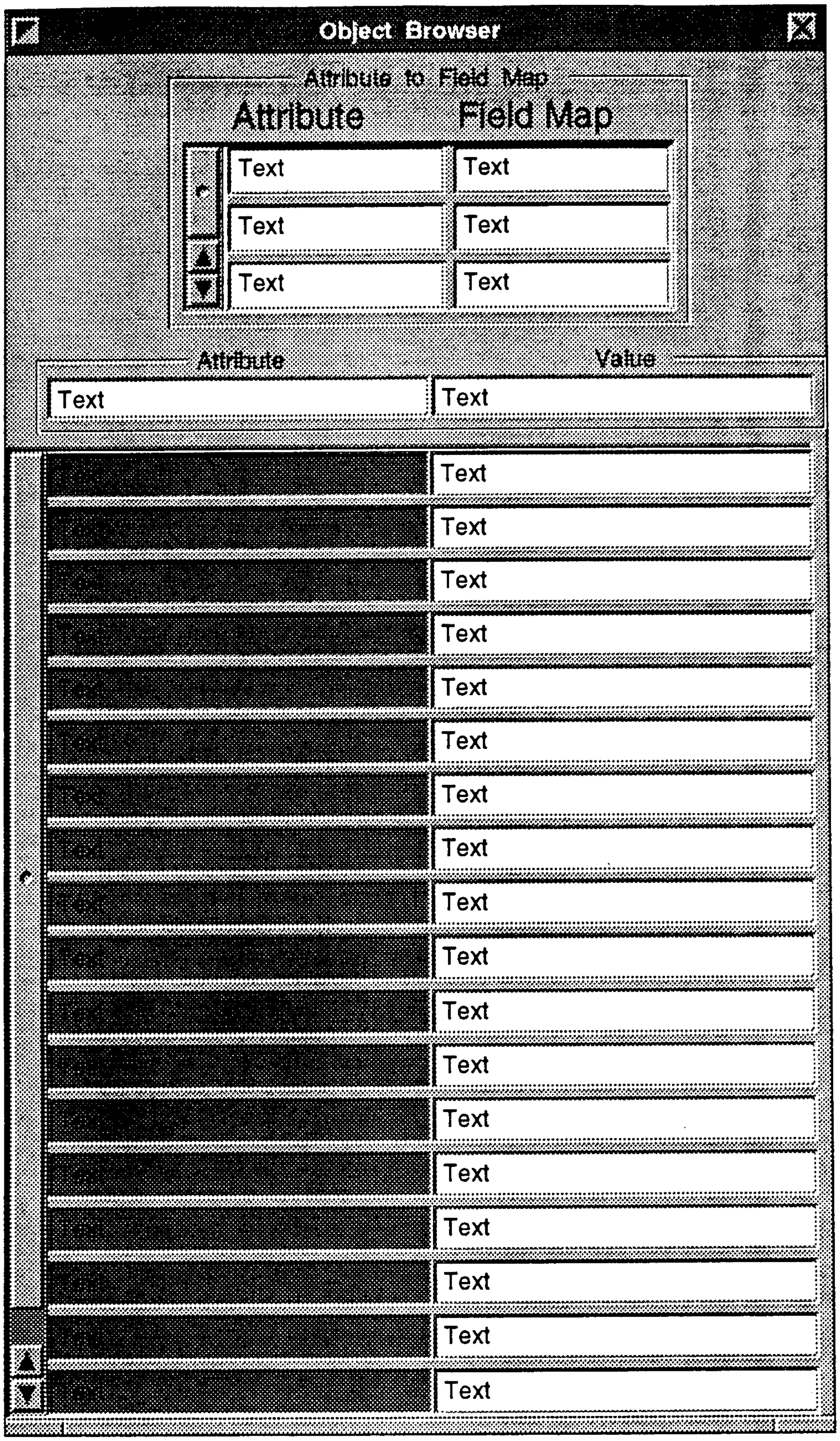


Figure 4-7: Object Browser.

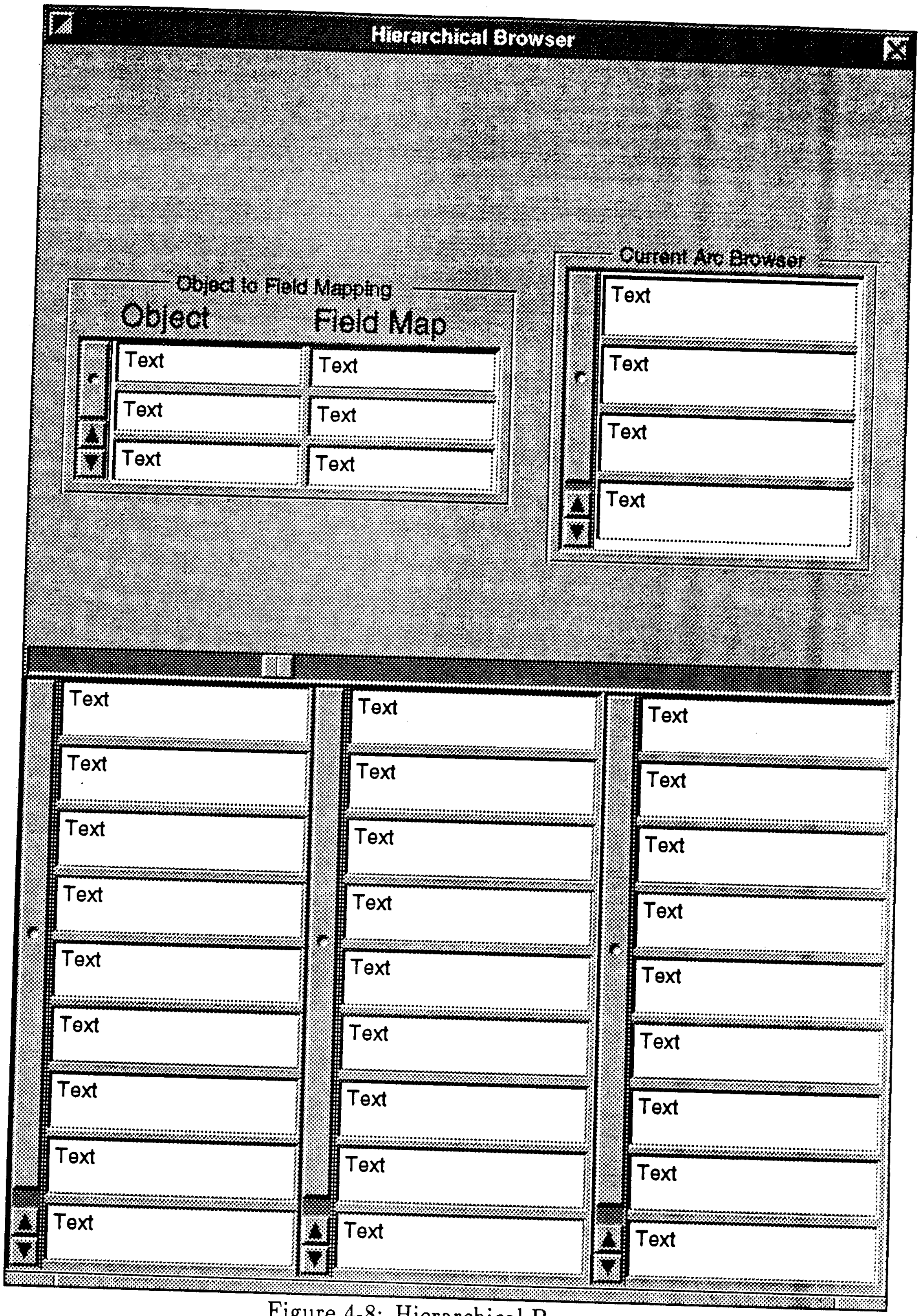


Figure 4-8: Hierarchical Browser.

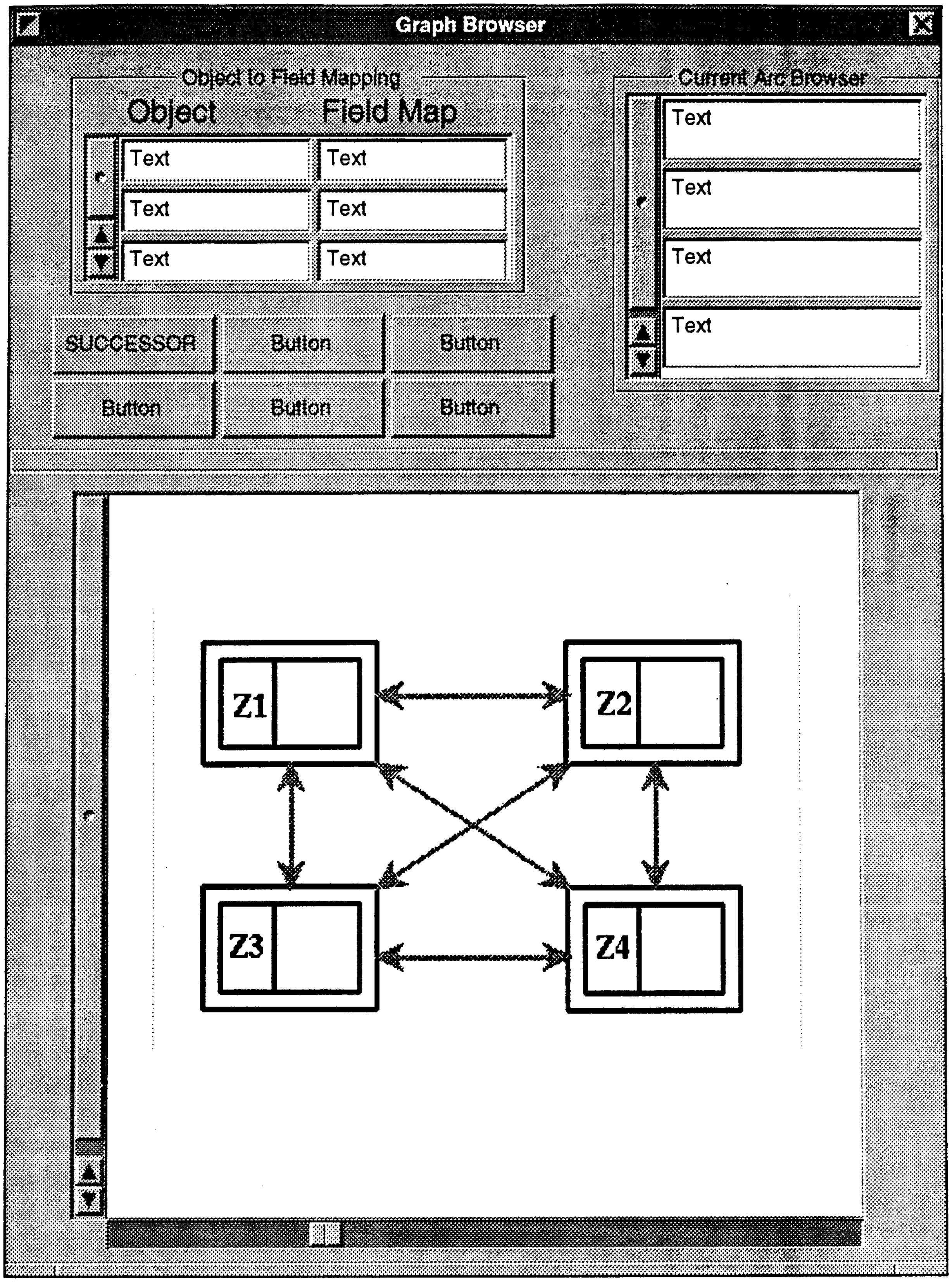


Figure 4-9: Graph Browser

```
Display Command Buffer Associated with window

Objects displayed in window

Object-View-Mapping
Object
Position Mapping
Value Mapping Table
Value Mapping Procedure
```

4.14 Interface Context

There are several values which are used to describe the current state of the interface.

- list of windows
- list of browsers
- links between windows and browsers
- current window
- current browser
- current display mapping
- current display mapping table
- the current object resulting from a browse or display action.

4.15 Interface Command Language

Messages can be sent to displays interactively through the command buffer.

All of the functionality of the interface is accessible through an interactive command language. The interface command language describes the overall functionality of the interface. All the interactive commands can also be placed in code for developing reusable scripts. The Command language supports defaults during interaction for conciseness and brevity.

4.15.1 Command Syntax

This is an initial stick in the ground. It comes from the display commands used in KBVision and PowerVision. The general syntax is

```
Display-action Window-Identifier object-set [keyword-arguments]*
              Snapshot File

Browse-action Browser-Identifier object-set [keyword-arguments]*
```

4.15.2 Window Creation

- Screen Size
- Screen Position

There are defaults maintained in global parameters describing the environment that are used if these are not specified. These can be changed from grabbing a window and resizing it in a manner consistent with the GUI package that is used.

`[cw` [1] `:size 512 512 :position 100 100]`

If a display operation is performed with no existing window, one is automatically created. Otherwise the *Current-Window* is used.

4.15.3 Browser Creation

4.15.4 Animation Commands

4.15.5 Window Linking

Display Windows can be Linked. A link between windows specifies a concatenation of functions to be applied to the domain and the range of the spatial object for mapping onto a display window. An example is using the display in one window to zoom onto a selected area in another window. In this case, the mapping from the domain of the object onto window 1 is concatenated with the mapping from the object onto window 2. Whenever a display mapping occurs in a window all of the associated linked windows are updates. For simplicity, displays are only updated when a display action is performed, not when changes are made to the spatial object. Cycles in window links are not allowed and therefore should be discouraged.

[1] See Section 4.15.9 on page 71 for a list of Interface Command Language Abbreviations

The basic commands for links are:

```
[create-link w1 w
     {followed by any of the keywords used to specify viewing an object}]

[unlink w1 w2]

[set-link w1 w2
     {followed by any of the keyword used to specify viewing an object}]
```

For example,

```
[create-link w1 w2 :zoom 2 2 :pan 100 100]
```

creates a link between window w1 and w2 such that whatever is displayed in w1 will appear in w2, zoomed by a factor 2 and panned by (100,100).

```
[cw :rgb-24]
> *W1*
[cw :copy-attributes *W1*]
> *W2*
[cw :copy-attributes *W1*]
> *W3*
[cw :copy-attributes *W1*]
> *W4*
[link w1 w2 :value red-component :red-8]
[link w1 w3 :value green-component :green-8]
[link w1 w4 :value blue-component :blue-8]
[p *W1* color-image]
```

Will display the different components of a color image in the three different windows.

(**Note**: Can windows can be linked to themselves (as in zooming into the same window)?)

Set-link changes the specified transformation between as also does a redisplay in the in the child window. For example,

```
[create-link w1 w2 :zoom 1 1]

[create-gizmo :type slider :range -5 5 :name zoom-slider
     :action [set-link w1 w2 :zoom zoom-slider.value zoom-slider.value]]
```

creates a link between two windows and a slider gizmo. When the slider gizmo is changed, it specifies how much zooming should occur and a redisplay occurs in w2 (the mapping from the spatial object onto w1 is not changed).

(**Note**: in interactive versions for zooming and panning should the selection square should be an object, like a cursor?)

4.15.6 Object Position Mapping Keywords

- zoom
- pan
- affine mapping
- viewing coordinate transforms
- matrix
- position-function

4.15.7 Graphics

Often times the user will want to perform common graphical display for things such as text, simple two dimensional graphics, more complicated three dimensional graphics. Examples are annotating a display, indicating where some action is occurring (the position of an epipolar line, translational flow paths, etc.), projecting a wireframe of a model onto an image.

This functionality can be found in several existing graphics packages and it would be best if we could just link to such a package. An alternative is to come up with thousands of commands like

```
[draw-line 100 100 200 200 :color blue :thickness 3]
[draw-circle ....]
```

for circles, disks, text, and so forth, which will be specific to the IUE. This may be required if such graphics packages are not accessible in an interactive form for our interface.

It is important to distinguish three different modes in which graphic displays can take place:

- they can occur in the coordinate system of the display window. In this case displays only occur with respect to the window coordinate system.
- they can occur in the coordinate system of the displayed object. In this case I would be drawing a line with respect to the inverse mapping from window to object coordinates.
- when the display is performed with respect to the coordinate system of an object, it can actually generate an instance of an IUE object. Thus, in drawing a line in object coordinates, an instance of an IUE line object would actually be produced. When the wireframe model is displayed, each line-segment and junction will be created as an object in the IUE. For composite objects this may require significant processing. It is straight forward for simple objects such as polygons, curves, and so forth. This is very useful for producing data for testing routines. This mode can be coupled with the interactive processing mode to allow for the interaction creation of data.

these different modes could be specified by a global mode or as keywords in the commands.

4.15.8 Object Value Mapping Keywords

These are for specifying commonly used transformation from an object value onto a screen intensity or color:

- overlay-color. Can take values such as red, blue, green, clear (clear the overlay plane), cycle (cycle through a set of specified overlay colors so that features displayed at different times get different colors: will select the next available overlay color),user defined colors. If no value is specified, it means the display will take place in the overlay plane either using a default or as specified by a value-function. The overlay plane can be thought of as a glass sheet on which the user can write annotations and display extracted features and values. For a displayed image, the overlay will occur in register with the display window. For a displayed surface or volume, the overlay can occur with respect to the display window OR with respect to the surface or volume (**Note:** We need a way of specifying this).

- value-function: User can specify an arbitrary function which takes in an object value and returns a value to be displayed in a display window in terms of intensity or color. This is a little chunk of code that the display process uses when it decides how to color a value from the object. For example,

```
[:overlay-color (red, green, blue, violet)]
[p image :value-function [if (image.value > 10) red blue]]
```

  The first command tells the current display to change the display mapping table to use the specified overlay colors (in the display toolbox there will certainly be an interactive color editor for selecting and creating overlay colors). The second will display red in the overlay plane at a screen pixel corresponding to an image pixel if the image value is greater than 10, otherwise it will display blue.

```
[p label-image :value-function [if (label-image.value = NULL)
                                   0
                                  (length label-image.value)]
               :linear 0 20 0 *screen-max*]
```

  This function displays a label image (an image where each pixel contains a list of all the objects which occupy that pixel). The value function determines the number of objects in this list and the linear function maps this onto available screen intensities. The object value mappings are applied in the order of the keywords.

- **overlay-object** the object which will be overlaid. This is generally used when one object determined where something will be displayed and another determines what color it should be. For example, suppose I have a binary edge image that I want to overlay on top of an color image:

```
[p image]
[p edge-image :overlay-color red]
```

  this could be specified as:

```
[p image :overlay-object edge-image :overlay-color red]
```

A typical use of this is for displaying extracted features with respect to a surface. Suppose I have a surface display of the intensity values in an image and I want to display the locations of an extracted edge-image on top of this:

```
[s image :overlay-object edge-image :overlay-color red]
```

- CLUT: User specifies a color look up table to map display object values through. This can be an array of numbers or an array of functions (pointers to functions).
- transparency operators: (**Note**: Consider implementations on NeXT and Silicon Graphics)
- linear: User specifies a range of values in the object and the range of screen value that they will be map on to define a linear function.

  ```
  [p image :linear 0.0 256.0 screen-min screen-max]
  ```

  would display image in the current window and linearly map object values from 0 to 256 onto the range specified by the system globals screen-min and screen-max (these would probably be defaults)

  ```
  [p image :linear 0.0 256.0 CLUT[1] CLUT[2]]
  ```

  map between values found in the specified color look-up table

4.15.9 Interface Command Language Abbreviations

p for pixel

i for interactive

v for vector

b for browse

cw for create window

cb for create browser

g for graphics

s for surface

4.15.10 Interface Context and Intelligent Defaulting

4.15.11 Organization of System Menus

4.16 Implementation of Nice Features/Examples

This section shows how nice features found in other IUE system will be implemented in the IUEUI by using it's basic objects. It also presents examples for typical or interesting interface operations.

4.16.1 Displaying an Image

The simplest way of doing this is to hit the display button on a display window or to type the command

```
[p]
```

and the *current object* will be displayed in the *current window* using the *current object display mapping* and the links to other windows and browsers associated with the *current window*.

A different current window can be selected by clicking anywhere on a window. The *current object* can be selected interactively from a browser. For example the command

```
[browse Objects-from-session.September20]
```

would either create or use the *Current-Browser* on all the objects saved in a database from a session on September20. The user would then use the command buttons associated with the browser to select an object as the *current object* and then type

```
[p]
```

The selected object will be displayed using a default *object display-mapping*. If no window exists, one will automatically be created based upon the object and system defaults.

The user can play with the *object display mapping* using keywords. For example,

```
[p :CLUT clut51]
```

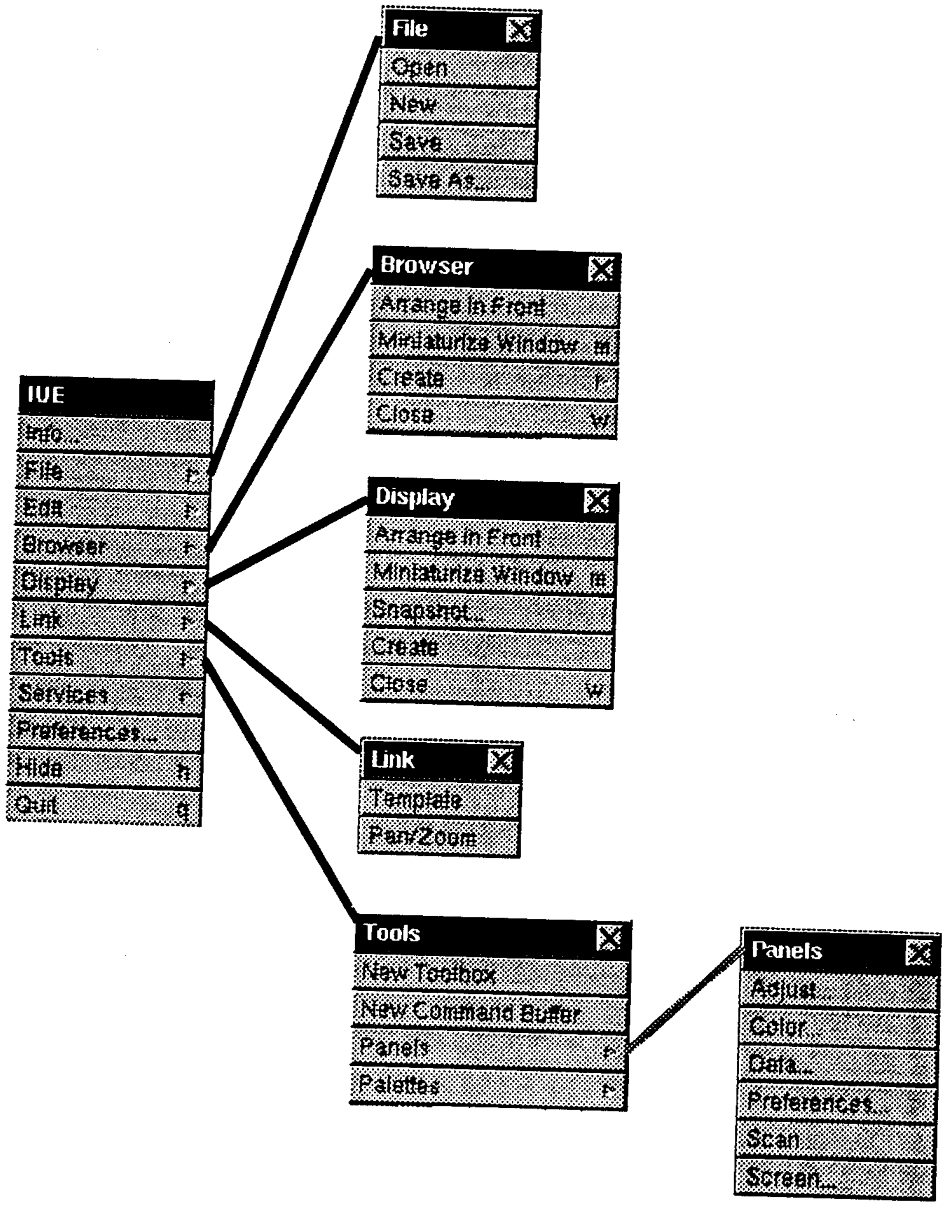


Figure 4-10: System Menu Layout

would display using the current defaults while the user supplies the color look-up table to map from object values to screen values.

```
[p :linear 0 128 *screen-min* *screen-max*]
```

would display using the current defaults while the range of object values from 0 to 128 are linearly mapped onto the range of values *screen-min* and *screen-max* (these could be defaults).

Additionally the user could type

```
[create-gizmo :type slider
              :name slick
              :min 0 :max 256
              :action
       [p image :linear 0 [read slick] *screen-min* *screen-max*]]
```

4.16.2 Interactive Function Application

The interactive display allows a user to access the currently displayed object or a list of specified objects through a display window using a mouse, pointing device or other interactive device (space ball, data glove). We begin with the simple form of this where we assume the user has a mouse and a keyboard. The command [This may belong as a menu-level command or a default when clicking in the window]

```
[i]
```

places the user in interactive mode with respect the *current window* and the *current object*. Thereafter, when the user clicks in locations in the window, he is returned the corresponding locations and and values in the displayed spatial object. These values can be displayed in the Command Interface or in the small text field associated with the current window. What is actually happening each time a click occurs in the window, is that the queues for *object-positions*, *object-values*, *window-positions*, *window-values* are being filled with the selected values.

```
          [i image1 image2 image3]
```

will do this for all the specified images simultaneously and store lists of values in the queues. To exit interactive mode, the user will hit some specified mouse button or keyboard key. (**Note**: what about viewing objects such as closed surfaces which may be two sided? We assume that the IUE will provide basic operation for intersecting a ray of projection with a set of specified objects with respect to a coordinate system. This operation could be invoked from the values returned by the i command. In general, all the geometric operations should be general ones to the IUE which can be invoked through the interface.)

The user can also call functions in the interactive mode to be applied to the values in the different queues. For example:

```
[i image  :1 [p :overlay-plane clear]
             [p image :value-function '(if (image.value > *object-value*[1])
                                           red blue)]
```

Whenever the user hits the terminal key 1, the overlay plane will be cleared and all image locations with a value greater than the value at the selected image location will be displayed in red, otherwise blue, in the overlay planes. *image.value* is a dummy variable that refers to the current value in image which is being displayed. **object-value*[1]* refers to the value selected using a mouse click in the display window. *red blue* refers to globally defined overlay colors. Recall that the *:value function* specifies the operation to be applied to an object value to map it onto a screen intensity or color **Note**: should there be a default for transparency operations?). Values in image are returned whenever the user clicks the mouse.

Another example:

```
[p *W1* image1]
[cw :copy-attributes *W2*]
[p *W2* image2]
[i *W1* image1 :1 [g *W2* :draw-line
                            *W1.min-x*
                            *object location*[1].y
                            *W2.max-x*
                            *object-location*[1].y]]
```

The user displays image1 in *W1*. He then creates another window which is identical to *W2* and interactively moves it to another screen position. He displays image2 in this new window using the *current-object-display-mapping*. Thereafter, when he interactively selects a point in image1, the corresponding epipolar line (no rotation, only translation in X) is displayed with respect to image2.

Let's say the user has a label image containing a set of extracted image features and he want to inspect these in detail.

```
[cw :size 512 512 :name zoomie]
> *W1*
[cw :size 128 128 :name zoomer]
> *W2*
[create-link *W1* *W2* :transformation-link :zoom 5 5 :pan 0 0]
[cb :type object :field-number 5]
[p *W1* image]
[p curveDB :positions curveDB.locations :overlay red]

[i label-image :always [browse *object-value*]
                       [set-link *W1* *W2* :pan *object-position*[1].x
                                                *object-position*[1].y]]
```

Here the user creates two windows and displays the image and the extracted database of curves in one of them. This is then displayed in *W2* under the specified transformation. The user creates an object browser. When the user clicks in *W1* with respect to the label plane, the curve object at that location is return as the *object-value*. This is then browsed in the current browser and the area surrounding this curve is displayed in *W2* to get a close up on the curve and the underlying image.

```
[i image
   :initial [message "select endpoints and press 1"]
   :1       line-mask = [make-instance
                                 :object line
                                 :domain-increment 1
                                 :interpolation nearest-neighbor
                                 :pt1 *object-position*[2]
                                 :pt2 *object-position*[1]]
   [p :overlay clear]
```

```
[p line-mask :positions line-mask.locations :overlay-plane red]
[plot1d *W2* :y-values [object-compose line-mask.positions image]]
```

Here the user is interactively creating a line-mask with mouse action, displaying the line-mask with respect to the image, and then plotting the image values found along the line-mask in as a 1d plot. Many things in this example are operations in the IUE that are still being specified. `line-mask = [make-instance :object line` creates an instance of a line object with integer indices that are mapped onto the image coordinates between pt1 and pt2 using nearest-neighbor interpolation. Such a functionality will exist in the IUE. There is also the issue of how we handle local variables (such as line- mask) in the IUEUI command buffer. Once the line-mask is created, it is displayed in the overlay plane. Finally, the composition of the positions in the line line-mask with the image (basically composing the functions line-mask: integers $-/>$ 2D Image Coordinates and image: 2D Image Coordinates $->$ scalar values) are plotted.

4.16.3 Displaying a set of Region Boundaries and Junctions

The user would type

```
[p image]
```

to display the image from which the features were extracted. RegionDB and JunctionDB refer to a database of extracted regions and junctions. To display these the user could type

```
[p RegionDB.locations :overlay cycle]
[p JunctionDB.locations :overlay green]
```

This would display the regions by cycling through the different defined overlay colors and then displaying the junctions in the green overlay plane. There are problems with this: adjacent regions may have the same overlay color and green junctions may be overlaid on top of green regions. The adjacent region problem could require a region coloring option (or the user could display the region contours in an overlay plane). The other problem could be solved by using compositing options such as exclusive-oring or by commands such as

```
[p RegionDB.locations :overlay cycle (red,blue,yellow,pink,orange) ]
[p JunctionDB.locations :overlay green]
```

where the set of overlay values to cycle through is explicitly listed (and doesn't include green). (**Note**: another possibility is to animate the overlays so they flash using double buffering animation).

RegionDB.locations reflects some general issue with how to refer to IUE objects and their attributes during display operations. First, we are referring not to a single object but to a set of objects that we want displayed. The display method will have to iterate over the elements of this set applying the appropriate display method to each. Second, regions and junctions have several attributes and we need to be clear about exactly what we want displayed from them. Here we are referring to the locations in image coordinates that are occupied by these objects. Let's say we wanted to color the regions based upon some attribute such as the value of some associated texture measure (what will these references look like in the IUE?). We suggest something like

```
[p RegionDB :positions RegionDB.locations :values
RegionDB.texture-measure
            :linear 0 100 *min* *max* :red-8]
```

This says to display the RegionDB with the positions coming from the locations attribute of the regions in the regionsDB and the values by taking the Region DB texture mappings and using a linear mapping from these onto screen intensities in 8 bits of red.

or

```
[p RegionDB :positions locations :values texture-measure
            :linear-mapping 0 100 *min* *max* :red-8]
```

Where it is locations and texture-measure are defined attributes of region objects in the regionDB.

4.16.4 Display of a Vector Field

```
[p image]
[v image.gradient :x-inc 5 :y-inc 5 :length-scale 5]
```

each component is being used to specify the components of the displayed vector. This should be a general operation where the components are used to describe the display of an arbitrary graphical object, such as a perspective display of a cube to provide information about orientation

4.16.5 Window to window Zooming and Panning

Creation of windows and browsers and creating links between these is a basic processing mode and can be performed through system menus. The user would select create-window from the system menu and create two windows by clicking and dragging the mouse to size the windows and would then position the windows by dragging the menu-bars. He could select the [either off a menu or from a button in the title bar of the window] the interactive window tools/inspector to set attributes such as name, title bar color.He would then select link-windows from the system menu. The order in which he then clicks on the windows creates a link between them. A browser of transformation between windows comes up that he can select from. He would select zoom and pan and specify it using a dynamic selection rectangle. Thereafter, whenever a display occurs in window *W1* the same display occurs in window *W2* with the transformation specified by the link.

Alternatively, this could all be done through interface commands:

```
[cw :screen-position 100 100 :screen-size 256 256 :name "big" :bar-color red]
> *W1*
[cw :screen-position 100 300 :screen-size 256 256 :name "zoomer"]
> *W2*
[link *W1* *W2* :zoom 2 2 :pan 100 100]

[i  :1 [set-link *W1* *W2* :zoom {add (1 1) to current zoom value}]
    :2 [set-link *W1* *W2* :zoom {subtract (1 1) to current zoom value}]
    :3 [set-link *W1* *W2* :pan object.location[1].x object.location[1].y]]
```

4.16.6 Browsing a selected Image Area

The simplest way to do this would be to type

```
        [array-browse]
```

and an array browser will be set up on the *current object* at a default location and neighborhood size (probably the center and 10) using the *current object browse mapping* [what type of font is used, etc.] The dimension of the array is based upon the dimension of the object. The user could use the command buttons on the array browser to move the inspection point through the object (1D or 2D arrow pairs plus another for selecting dimension—the default setting would come from the browse method associated with the object to be browsed). Or the user could type

```
        [array-browse object]
```

if the user types

```
        [array-browse object1 object2 ...]
```

the fields in the array browsers will display registered values from all the specified objects. If the user types

```
        [array-browse object1 [Function1 object1] [Function2 object1]]
```

the fields in the array browsers will display registered values from object1 and the application of Functions 1 and 2 to object 1. If I have two different routines for computing curvature along a curve and I want to inspect them, I could type

```
        [array-browse curve [curvature1 curve] [curvature2 curve]]
```

An alternative is to run the array browser on mouse selected locations. This may be a default action relative to the current window or the user may type

```
        [i :continuous [array-browse
                          :location *current- object.x* *current-object.y*]]
```

and the browser will display values in a neighborhood surrounding the current object position selected by the mouse. The neighborhood will be updated continuously as the mouse is moved.

Setting up links between array browsers: the links have associated processing actions to see the effect of a routine over a selected neighborhood.

4.16.7 Displaying a Color Image

```
[p color-image :color rgb-8]
```

displays the color image in with 8 bits in red, green and blue.

```
[p image1 image2 image3 :color rgb-8]
```

treats the three images as specifying the rgb components

4.16.8 Multiple Image Display

```
[p image1 image2 :values [- image1.val image2.val]
        :linear-range -20 20 *min* *max*]
```

4.16.9 Interactive Inspection of Image Registered Features

This involves how spatial queries are performed in the IUE. One way involves the use of a label image. This is essentially a depth buffer which stores a a list of objects which occupy a given pixel (the list could be ordered by depth or time of extraction of feature). This can be treated as an image and all the operations that can be applied to an image can be applied to it. Such as

```
[p label-image :values [if (label-image.value  /= NULL)
[i label-image :1 [browse label-image.value]]
```

4.16.10 Finding a particular result and Displaying it

The user would browse a particular database by typing

```
[browse long-term-data-base
        :attributes (time-of-creation,function,type)]
```

and a browser would come up with respect to the objects in the long term data base listed row by row with the attributes time-of-creation, function, type in the columns. The user would click the column head **type** and then = and then type in "image" and the browser would then display all the objects of type "image". The user would then click the column head **function** and the = and then type in "Gaussian" and the browser would then display all the images created by the Gaussian Routine. The user would then click the column head Time-of-creation and then Sort Max and the browser displays the selected images ordered by their time of creation. By clicking to the side of the first listed image, it then becomes the *current-object* and can be displayed either by clicking the display button in the browser or by typing

```
        [p]
```

Suppose the user had a database of regions and wanted to find the largest. He would type

```
        [browse RegionDB :attributes area ]
```

4.16.11 Animation of Surface Display of a moving edge

Here we are trying to understand the nature of an edge in a sequence of images. To do this, we display the intensity in a selected image area as a surface plot and then display, as an overlay, on top of the surface plot, the extracted edges. This is done using a sequence of images and extracted edge-images (binary images where 1 indicates the presence of an edge and 0 the absence) and writing the display out to an animation file and then playing it back in a window.

To do this, the user would type:

```
{iterate image over image-sequence and
         edge-image over edge-image-sequence

        [s animation-file image :image-position 200 200
                                :distance 100
                                :Theta1 20
                                :Theta2 3
            :overlay-object edge-image :overlay-plane red :clear t]}

[play animation-file]
```

Several things are occurring here. The first two lines correspond to the set iteration control structures provided in the base programming language. We don't want to build a complete programming language into the interface command language so it's not clear if this is interactive. These lines cause the variables image and edge-image to iterate over the respective sets. The next line says to do a surface display to an animation file. Instead of displaying in a window directly, the display will be written out to a file for play back. If a window were specified instead, the display would occur in a window (although this may not play at an effective speed for an animation). The surface displays selects an image point location which will be viewed, viewing distance from this point and the angles of view. It also states that any previous display actions will be cleared (it's a new frame) and that the edge-image is displayed in red values at locations it occupies on top of the intensity surface.

The play command will play the animation back in the current window. There are keywords associated with the play command for controlling the speed of the animation. In fact, these parameters could be linked to a slider for interactive control.

Animations can be made on the fly. A user can always specify this by indicating an animation file instead of a display window in a display command. Another command is

```
[snapshot window animation-file]
```

which will store whatever the state of the display in the specified window is out to an animation file.

4.16.12 Interactive Surface Display

Here the user specifies a location in an image in one window and in another window, a surface plot around this location is displayed. The user would type:

```
[p w1 image]
[i w1 :1 [s w2 image :location image.x image.y
                     :distance 200
                     :theta1 30
                     :theta2
                     :clear t]]
```

another possibility would be

```
[create-gizmo :type slider :range 0 1000 :name distance]
[create-gizmo :type slider :range 0 180 :name theta1]
[create-gizmo :type slider :range 0 180 :name theta2]
[p w1 image]
[i w1 :1 [s w2 image :location image.x image.y
```

```
                                    :distance slider.distance
                                    :theta1 slider.theta1
                                    :theta2 slider.theta2
                                    :clear t]]
```

another possibility would be to allow variables that could be set by expressions in the interface.

4.16.13 Process Monitoring

This is essentially a Set/DataBase Browser on a database of task objects. To be KBV-ish, one of the field column should be mapped onto red or green. Tasks would change the colors in their corresponding fields based upon their status

4.16.14 Constraint Network Monitoring

This section needs to be created in coordination with the design of the IUE constraint networks.

4.16.15 Interactive Histogram Segmentation

Clicking on (or near the axis of a plotted function) returns the x coordinate and the y-value of the displayed object.

```
[plot2d *W1* histogram]
[p *W2* image]
[i *W1* histogram :1 [min = current-value[1].x]
                     [max = current-value[2].x]
                     [p *W2* image
                        :value-function
        [if ((image.value > min) & (image.value < max)) blue]]]
```

Here the user has plotted a histogram in *W1*. He then selects the range of values by clicking on the displayed histogram. The current-object-value contains the x and y value from the displayed histogram. These are stored in the local values min and max (this isn't necessary: in general how will we refer to local variables in these expressions? Will there be a set of dummy variables). When the user hits the key 2, the selected range of values are displayed in the blue overlay plane.

Section 5

Summary

The IUE will be a revolutionary system covering many technologies, serving many levels of users, and growing by distributed development over many years. The class hierarchy framework and user interface concepts presented in this document will allow:

- Integration of the diverse concepts of IU within one environment.
- Rapid introduction of new users to the IUE.
- Organized extension of the base IUE by developers (including new users).